



Universidad  
Carlos III de Madrid

## **PROYECTO FIN DE CARRERA**

*Escuela Politécnica Superior Ingeniería en Informática  
Departamento de Informática*

### **AdaptBenchmark:** *El benchmark adaptativo universal*

*AUTOR: Antonio Díaz Ponce  
TUTOR: David Expósito Singh*



**Título:** AdaptBenchmark: El benchmark adaptativo universal

**Autor:** Antonio Díaz Ponce

**Tutor:** David Expósito Singh

## **EL TRIBUNAL**

**Presidente:** Jesús Carretero Pérez

**Vocal:** Francisco Javier García Blas

**Secretario:** Alberto Valero Gómez

Realizado el acto de defensa y lectura del Proyecto Fin de Carrera el día 12 de diciembre de 2011 en Leganés, en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de: **MATRÍCULA DE HONOR**

VOCAL

SECRETARIO

PRESIDENTE

# AGRADECIMIENTOS

---

En primer lugar quiero agradecer y dedicar la creación de este proyecto a mis padres, por ser siempre un fuerte apoyo tanto en los buenos como en los malos momentos. A mi hermano, por no ser sólo un hermano, sino el mejor amigo que podría tener; a mi familia: tíos-tías y primos-primas, y en especial a mis abuelos y abuelas que seguro que estarán contentos de ello.

Quiero dar las gracias a mi tutor, David Expósito, por su gran aporte a este proyecto, por la cordialidad y el buen trato durante las tutorías, por esas ideas que a mí jamás se me ocurrirían. Y por aguantarme durante tanto tiempo. Ha sido un placer trabajar contigo David.

A mis compañeros de universidad y amigos: Álvaro y Javier, con vosotros he pasado el mayor tiempo haciendo esas interminables prácticas y con vosotros he vivido momentos que jamás olvidaré, sois muy grandes. Gracias a Lidia y Roberto por ser los primeros en dar sin recibir nada a cambio y por vuestra forma de ser, no cambiéis nunca. Gracias Christian por ser uno de los compañeros más eficientes que he conocido y por ofrecer ese punto de vista siempre desconocido. Gracias Óscar y Laura por demostrar que después de la universidad no acaba todo sino que comienzan mejores momentos. A Luis por ser un gran compañero, por ofrecer constantemente ayuda y consejo, y por su humor inteligente. Gracias Córdoba por ser guía, amigo y compañero de equipo durante toda la carrera.

También quiero agradecer a mis amigos de primera por conseguir que olvide los malos momentos con sólo pasar un rato con vosotros. Gracias por estar ahí siempre proponiendo nuevas cosas.

No podría faltar en este agradecimientos mis amigos del colegio: Luis, Lorenzo y Fernando; por seguir siendo igual desde hace tanto tiempo sin dejar que nada ni nadie os cambie, siempre habéis sido esa constante que todo el mundo tiene que tener para sentirse feliz.

Por último, y no menos importante dar gracias a todos los compañeros de equipo con los que he conseguido seguir aprendiendo nuevas cosas de vosotros. Milan gracias por todo aunque estés lejos, no te olvidaremos.

Gracias a todos los que aunque no haya nombrado se sientan identificados con parte de esto.

Un abrazo fuerte.

# RESUMEN

---

Desde la primera aparición de las plataformas de computación, ha sido necesaria la comparación del rendimiento de los distintos productos. El término benchmark emerge a finales del siglo XIX, definido como un punto de referencia en la toma de medidas. Actualmente, este tema está más relacionado con herramientas software que evalúan el rendimiento del ordenador. Este proyecto trata sobre la definición, creación y evaluación de un conjunto de benchmarks para tomar medidas del rendimiento de los microprocesadores. Lo hemos denominado AdaptBenchmark: el benchmark adaptativo universal. Este benchmark incluye diferentes test que evalúan varias características de la arquitectura de un procesador.

En este documento explicamos la metodología que hemos seguido durante el desarrollo del AdaptBenchmark. Detallamos la estructura interna del software, que ha sido concebida para evaluar el rendimiento de modernos procesadores con varios núcleos. AdaptBenchmark mide el tiempo de ejecución y eventos internos como: fallos caché en L1 y L2, tráfico de bus o fallos TLB. Esta información es recogida utilizando contadores hardware y resumida mediante gráficas.

Para cada test desarrollamos un estudio extensivo del desarrollo de procesador así como un análisis de las interacciones entre software y el correspondiente hardware. Además, introducimos una técnica para buscar la mejor configuración en las distintas plataformas. Para ello utilizaremos dos técnicas de optimización: Simulated Annealing y la Búsqueda N-aria. Esta última fue creada específicamente para este proyecto. Con la utilización de estas metodologías es posible adaptar el benchmark a distintos contextos hardware.

A lo largo de este proyecto, trabajaremos con dos arquitecturas diferentes: Intel Core2Duo y AMD Opteron. Y compararemos al detalle el funcionamiento de estas dos plataformas para distintas configuraciones, probando así la eficiencia de nuestra herramienta.

Palabras clave: benchmark, rendimiento, evaluación, arquitectura, microprocesador.

# ABSTRACT

---

Since the first appearance of computing platforms, it was necessary to compare the performance of different products. The term benchmark emerges at the end of 19<sup>th</sup> century, defined as a point of reference for a measurement. Nowadays, this topic is commonly related to software tools that evaluate the computer performance. This project is about the definition, creation and evaluation of a set of benchmarks for measuring the microprocessor performance. We call them AdaptBenchmark: a universal adaptive benchmark. This benchmark includes different tests that evaluate several characteristics of the processor architecture.

In this document we introduce the methodology that we have followed during the development of AdaptBenchmark. We detail the internal structure of this software, which was conceived for evaluating the performance of modern multicore processors. AdaptBenchmark measures the execution time as well as internal processor events like L1 and L2 cache misses, bus traffic or TLB misses. This information is collected using the hardware counters and then summarized in chars.

For each test we perform an extensive study of the processor performance as well as an analysis of the interactions between the software and the underlining hardware. In addition, we introduce a technique for fine-tuning the benchmark in order to find the best configuration for each platform. In order to do this, we use two optimization techniques: the Simmulated Annealing and the n-ary Search. The latter one was originally created in this project. With the use of this methodology it is possible to adapt the benchmark to different hardware contexts.

Finally, we perform the evaluation of AdaptBenchmark for two different architectures: Intel Core2Duo and AMD Opteron. We make a detailed comparison between these two platforms for different hardware configurations, proving the efficiency of our tool.

Keywords: benchmark, performance, evaluation, architecture, microprocessor.

Caminante, no hay camino, se hace camino al andar.

Antonio Machado

# ÍNDICE GENERAL

---

ÍNDICE GENERAL.....	8
1.- Introducción .....	13
1.1 Motivación general .....	13
1.2 Objetivos del proyecto .....	14
1.3 Introducción de la memoria.....	16
2.-Plataforma de evaluación .....	17
2.1 Intel Core 2 Duo T7100 .....	17
2.2 AMD Opteron 6168 .....	20
2.3 Contadores hardware .....	22
2.4 Entorno software .....	25
2.4.1 OpenMP .....	25
2.4.2 Librería PAPI .....	26
2.4.3 Prefetching .....	27
2.4.3.1 Prefetching Hardware, MSR-TOOLS.....	28
2.4.3.2 Prefetching Software.....	31
2.4.4 Huge Pages .....	31
3.- Benchmarks de evaluación.....	33
3.1 Escritura con stride.....	33
3.2 False sharing .....	36
3.3 Producto matriz-dispersa vector .....	39
3.4 Técnica de optimización .....	43
3.4.1 Simulated Annealing .....	43
3.4.2 Búsqueda N-aria .....	46
4.- Experimentos .....	49
4.1 Definición de parámetros de la arquitectura a evaluar .....	49
4.1.1 Selección dinámica de contadores .....	50
4.2 Metodología para evaluar los parámetros de la arquitectura.....	52
4.3 Metodología de análisis de los resultados .....	54
4.4 Escritura con stride.....	57
4.4.1 Intel Core2Duo .....	57
4.4.1.1 Prefetching .....	61
4.4.2 AMD Opteron .....	64



4.4.3 Comparativa .....	66
4.5 False sharing .....	69
4.5.1 Intel Core2Duo .....	69
4.5.1.1 Prefetching .....	71
4.5.2 AMD Opteron .....	73
4.5.3 Comparativa .....	76
4.6 Producto Matriz-dispersa vector .....	78
4.6.1 Dispersión .....	79
4.6.1.1 Intel Core2Duo.....	81
4.6.1.2 AMD Opteron .....	84
4.6.1.3 Comparativa .....	85
4.6.2 Número de hilos .....	87
4.6.2.1 Intel Core2Duo.....	87
4.6.2.2 AMD Opteron .....	90
4.6.2.3 Comparativa .....	91
4.6.4 Tipos de planificación.....	96
4.7 Huge pages .....	98
4.8 Integración con Técnicas de optimización.....	100
4.8.1 Simulated Annealing .....	101
4.8.2 Búsqueda N-aria .....	107
4.8.3 Comparativa .....	109
5.- Comparativa Final.....	112
6.- Planificación y presupuesto.....	116
6.1 Planificación.....	116
6.2 Presupuesto .....	120
7.- Conclusiones .....	123
7.1 Líneas futuras.....	124
8.- Glosario.....	126
9.- Referencias.....	127
Apéndice A: PAPI Guía de instalación y uso .....	130
Apéndice B: Ejemplo de uso de Huge Pages.....	159
Apéndice C: Manual de usuario.....	160

# ÍNDICE DE FIGURAS

Figura 1: Imagen renderizada con 3DMark .....	13
Figura 2: Funciones de correspondencia directa y asociativa por conjuntos de 2 vías. ....	18
Figura 3: Arquitectura del procesador Intel Core 2 Duo. ....	18
Figura 4: Gráfica de situación de la caché L2 en un Core 2 Duo .....	19
Figura 5: Vista ampliada de un Opteron Istanbul .....	21
Figura 6: Pirámide de tipos de memoria en relación con CPU .....	23
Figura 7: Captura del renderizador de imágenes médicas AMIDE. ....	28
Figura 8: Captura en la que la BIOS permite desactivar el prefetching hardware .....	29
Figura 9: Prefetchers dentro del Intel Core 2 Duo .....	30
Figura 10: Prefetchers dentro del Intel Core 2 Duo .....	31
Figura 11: Acceso a matrices por filas y por columnas .....	33
Figura 12: Vista física de acceso por filas y por columnas .....	34
Figura 13: Ocupación de los núcleos en la prueba escritura con stride .....	35
Figura 14: Ejemplo de reparto de tareas en false sharing .....	36
Figura 15: Ejemplo1 traza false sharing .....	37
Figura 16: Ejemplo2 traza false sharing .....	37
Figura 17: Ocupación de los núcleos en la prueba false sharing .....	38
Figura 18: Producto entre una matriz y vector .....	39
Figura 19: Matriz dispersa, comprimida triplete y CSR .....	39
Figura 20: Matrices con grados de dispersión 1, 2, 5, 10 y 20 respectivamente, número de filas 5000, número de elementos por fila 500 .....	41
Figura 21: Reparto de la carga de producto de matriz-vector por filas y columnas .....	42
Figura 22: Ocupación de los núcleos en la prueba producto matriz-dispersa vector. ....	42
Figura 23: Fórmula de cálculo de probabilidad en Simulated Annealing .....	43
Figura 24: Fórmula de la constante Boltzmann .....	44
Figura 25: Explicación de Simulated Annealing .....	44
Figura 26: Ejemplo de Búsqueda N-aria .....	47
Figura 27: Diagrama de Búsqueda N-aria .....	48
Figura 28: Single data phase transaction vs burst transfer .....	50
Figura 29: Diagrama de funcionamiento del testeador de evento .....	51
Figura 30: Tiempos para experimentos 1 y 2 [ES][Core2Duo] .....	57
Figura 31: Caso en el que dos filas puedan solapar en un mismo bloque .....	58
Figura 32: Salida del simulador de escritura con stride .....	59
Figuras 33 y 34: Fallos L1 y L2 para experimentos 1 y 2 [ES][Core2Duo] .....	59
Figuras 35 y 36: Transferencias de bus y fallos TLB para experimentos 1 y 2 [ES][Core2Duo] .....	60
Figura 37: Muestra de utilización en una iteración .....	60
Figura 38: Número de ciclos para experimentos 1 y 2 [ES][Core2Duo] .....	61
Figura 39: Tiempo registrado para experimentos 2, 3, 4 y 5 [ES][Core2Duo] .....	61
Figuras 40 y 41: Fallos L1 y L2 para los experimentos 2, 3, 4 y 5 [ES][Core2Duo] .....	62
Figuras 42 y 43: Transferencias de bus y fallos TLB para experimentos 2, 3, 4 y 5 [ES][Core2Duo] .....	63
Figura 44: Número de ciclos para experimentos 2, 3, 4 y 5 [ES][Core2Duo] .....	63
Figura 45: Tiempos para experimentos 6 y 7 [ES][Opteron] .....	64
Figuras 46 y 47: Fallos L1 y L2 para los experimentos 6 y 7 [ES][Opteron] .....	65
Figuras 48 y 49: Transferencias de bus y fallos TLB para experimentos 6 y 7 [ES][Opteron] .....	65
Figura 50: Número de ciclos para experimentos 6 y 7 [ES][Opteron] .....	66
Figura 51: Tiempos para experimentos 2 y 6 [ES][Core2Duo y Opteron] .....	66
Figuras 52 y 53: Fallos L1 y L2 para los experimentos 2 y 6 [ES][Core2Duo y Opteron] .....	67
Figuras 54 y 55: Fallos TLB y número de ciclos para experimentos 2 y 6 [ES][Core2Duo y Opteron] .....	67
Figura 56: Tiempos para experimentos 8 y 9 [FS][Core2Duo] .....	69
Figuras 57 y 58: Fallos L1 y L2 para experimentos 8 y 9 [FS][Core2Duo] .....	69
Figuras 59 y 60: Transferencias de y fallos TLB para experimentos 8 y 9 [FS][Core2Duo] .....	70
Figura 61: Número de ciclos para experimentos 8 y 9 [FS][Core2Duo] .....	71
Figura 62: Tiempo registrado para experimentos 9, 10, 11 y 12 [FS][Core2Duo] .....	71
Figuras 63 y 64: Fallos L1 y L2 para experimentos 9, 10, 11 y 12 [FS][Core2Duo] .....	72

Figuras 65 y 66: Transferencias de bus y fallos TLB para experimentos 9, 10, 11 y 12 [FS][Core2Duo] ...	72
Figura 67: Número de ciclos para experimentos 9, 10, 11 y 12 [FS][Core2Duo].....	73
Figura 68: Tiempo registrado para experimentos 13 y 14 [FS][Opteron] .....	73
Figuras 69 y 70: Fallos L1 y L2 para experimentos 13 y 14 [FS][Opteron] .....	74
Figuras 71 y 72: Peticiones al bus de memoria y fallos TLB para experimentos 13 y 14 [FS][Opteron].....	74
Figura 73: Número de ciclos para experimentos 13 y 14 [FS][Opteron] .....	75
Figura 74: Tiempo registrado para experimentos 9 y 13 [FS][Core2Duo y Opteron].....	76
Figuras 75 y 76: Fallos L1 y L2 para experimentos 9 y 13 [FS][Core2Duo y Opteron].....	76
Figura 77: Explicación comparativa fallos L1 en false sharing [FS][Core2Duo y Opteron] .....	77
Figuras 78 y 79: Fallos TLB y número de ciclos para experimentos 9 y 13 [FS][Core2Duo y Opteron] .....	77
Figura 80: Tiempos para el experimento 15' [PMV][Opteron] .....	79
Figura 81: Accesos iteración a matriz concentrada y dispersa .....	80
Figuras 82 y 83: Tiempos para el experimento 15 con/sin dispersión 1 [PMV][Core2Duo].....	81
Figuras 84 y 85: Fallos L1 para el experimento 15 con/sin dispersión 1 [PMV][Core2Duo] .....	82
Figuras 86 y 87: Fallos en L2 y transferencias de bus para el experimento 15 [PMV][Core2Duo].....	82
Figuras 88 y 89: Fallos en TLB y número de ciclos para el experimento 15 [PMV][Core2Duo].....	83
Figura 90: Tiempo para el experimentos 16 [PMV][Opteron] .....	84
Figuras 91 y 92: Fallos en L1 y L2 para el experimento 16 [PMV][Opteron].....	84
Figuras 93 y 94: Peticiones al bus de memoria y fallos TLB para el experimento 16 [PMV][Opteron] .....	85
Figura 95: Tiempos para los experimentos 15 y 16 [PMV][Core2Duo y Opteron] .....	85
Figuras 96 y 97: Fallos en L1 y L2 para los experimentos 15 y 16 [PMV][Core2Duo y Opteron] .....	86
Figuras 98 y 99: Fallos en TLB y número de ciclos para los experimentos 15 y 16 [PMV][Core2Duo y Opteron] .....	87
Figuras 100 y 101: Tiempos y fallos en L1 para el experimento 17 [PMV][Core2Duo] .....	88
Figuras 102 y 103: Fallos en L2 y transferencias de bus para el experimento 17 [PMV][Core2Duo].....	88
Figuras 104: Número de ciclos para el experimento 17 [PMV][Core2Duo] .....	89
Figura 105: Tiempos para el experimento 18 [PMV][Opteron] .....	90
Figuras 106 y 107: Fallos en L2 y transferencias de bus para el experimento 18 [PMV][Opteron] .....	90
Figura 108: Tiempos para experimentos 17 y 18 [PMV][Core2Duo y Opteron] .....	91
Figuras 109 y 110: Tiempos para el experimento 19 con/sin Diag2 [PMV][Opteron] .....	93
Figuras 111 y 112: Fallos en L1 y transferencias de bus para el experimento 19 [PMV][Opteron] .....	93
Figuras 113 y 114: Peticiones el bus de memoria y fallos TLB para el experimento 19 [PMV][Opteron] ...	94
Figura 115: Explicación a los fallos en TLB par tamaño de grano fino .....	95
Figura 116: Número de ciclos para el experimento 19 [PMV][Opteron] .....	95
Figura 117: Tiempo en segundos para el experimento 20 [PMV][Opteron] .....	96
Figura 118: Tiempo en segundos para los experimentos 9 y 21 [FS][Core2Duo].....	98
Figuras 119 y 120: Fallos en L1 y L2 para los experimentos 9 y 21 [FS][Core2Duo] .....	98
Figuras 121 y 122: Transferencias de bus y fallos en TLB para los experimentos 9 y 21 [FS][Core2Duo]...	99
Figura 123: Número de ciclos para los experimentos 9 y 21 [FS][Core2Duo] .....	99
Figura 124: Función creada a partir de 11 puntos.....	100
Figura 125: Gráficas seleccionadas para la experimentación [PMV][Core2Duo] .....	101
Figuras 126 y 127: Puntos testeados por Simulated Annealing, 1 y 2 iteraciones [PMV][Core2Duo] .....	102
Figuras 128 y 129: Puntos testeados por Simulated Annealing, 3 y 5 iteraciones [PMV] [Core2Duo] .....	102
Figuras 130 y 131: Puntos testeados por Simulated Annealing, 25 y 45 iteraciones [PMV][Core2Duo] .	103
Figuras 132 y 133: Puntos testeados por Simulated Annealing, 45 iteraciones [PMV][Opteron] .....	103
Figuras 134, 135 y 136: Gráficas generadas por Simulated Annealing con distintas configuraciones para fallos en L1 después de 45 iteraciones [PMV][Core2Duo] .....	104
Figuras 137, 138 y 139: Gráficas generadas por Simulated Annealing con distintas configuraciones para fallos en L1 después de 45 iteraciones [PMV][Opteron].....	104
Figura 140: Muestra de los tres test en la misma gráfica [PMV][Core2Duo] .....	104
Figura 141: Muestra de los tres test en la misma gráfica [PMV][Opteron].....	105
Figuras 142 y 143: Fallos en L2 y transferencias de bus con Simulated Annealing (3 tests) [PMV][Opteron] .....	105
Figuras 144 y 145: Fallos en L2 y transferencias de bus con Simulated Annealing (3 tests) [PMV][Core2Duo] .....	106
Figuras 146 y 147: Puntos testeados por la Búsqueda N-aria, 1 y 2 iteraciones [PMV][Opteron] .....	107
Figuras 148 y 149: Puntos testeados por la Búsqueda N-aria, 3 y 5 iteraciones [PMV][Opteron] .....	108
Figuras 150 y 151: Puntos testeados por la Búsqueda N-aria, 10 y 14 iteraciones [PMV][Opteron] .....	108

Figura 152: Fallos en L1 generados por la Búsqueda N-aria [PMV][Opteron].....	109
Figura 153: Fases del desarrollo del proyecto junto con las tutorías .....	117
Figura 154: Diagrama de versiones durante la fase de implementación .....	118
Figura 155: Diagrama de código generado .....	119
Figura 156: TagCloud realizado sobre la memoria.....	119
Figura 157: Captura de pantalla de la aplicación Perfometer.....	124

## ÍNDICE DE TABLAS

Tabla 1: Especificaciones técnicas de Intel Core de 2 Duo. ....	17
Tabla 2: Especificaciones técnicas de AMD Opteron.....	20
Tabla 3: Pruebas a lanzar con distintas configuraciones de prefetching .....	27
Tabla 4: Resumen de experimentos. ....	55
Tabla 5: Resumen de apartados.....	56
Tabla 6: Número de filas para llenar caché del Core2Duo. ....	60
Tabla 7: Comparativa de características entre Core2Duo y Opteron.....	66
Tabla 8: Cálculo de tamaño de vectores en producto matriz-vector. ....	81
Tabla 9: Diferencias de fallos L1 y L2, entre Core2Duo y Opteron. ....	86
Tabla 10: Matrices a utilizar para la comparativa. ....	92
Tabla 11: Iteraciones asignadas por hilo con planificación guided.....	97
Tabla 12: Iteraciones asignadas por hilo con planificación static ó dynamic .....	97
Tabla 13: Resultados de Simulated Annealing y Búsqueda N-aria.....	110
Tabla 14: Comparativa de tiempos escritura con stride entre Core2Duo y Opteron. ....	112
Tabla 15: Comparativa de tiempos false sharing entre Core2Duo y Opteron.....	113
Tabla 16: Comparativa de tiempos producto matriz vector entre Core2Duo y Opteron. ....	114
Tabla 17: Comparativa de tiempos en false sharing con/sin Huge Pages [Core2Duo] .....	115

## ÍNDICE DE CÓDIGO Y SALIDAS

Código 1: Escritura con stride.....	35
Código 2: False sharing .....	38
Código 3: Salida del testeador de contadores.....	51
Código 4: Salida de prueba simple .....	52
Código 5: Salida benchmark modo rows, prueba matriz normal.....	53
Código 6: Salida final benchmark, prueba producto matriz-dispersa vector .....	54
Código 7: Producto matriz-dispersa vector, versión creada por nosotros para que se realice por filas .....	78
Código 8: Producto matriz-dispersa vector, sin utilizar compartición de vector escritura.....	78
Código 9: Producto matriz-dispersa vector, sin utilizar la matriz dispersa (se usa un vector)....	79

# 1.- Introducción

En este apartado introductorio trataremos de mostrar escuetamente el mundo de los benchmarks, así como el propósito de este proyecto. En el último apartado se puede ver de forma más concreta, las partes de las que está compuesta la memoria.

## 1.1 Motivación general

El benchmarking es un concepto que se utiliza mucho en la administración de empresas, y que consiste en un proceso encargado de comparar productos o servicios entre distintas compañías, o dentro incluso de una misma corporación <sup>[1]</sup>. El término proviene del inglés y podría traducirse como medida de calidad o punto de referencia <sup>[2]</sup>.

Aplicado a la informática este proceso sirve para comparar el funcionamiento de dos productos, ya sean CPUs, tarjetas de memoria (RAM), tarjetas gráficas, discos duros o cualquier tipo de componente asociado a ellos <sup>[3]</sup>. Sin embargo, los principales benchmarks tratan únicamente el estudio del rendimiento del procesador. En nuestro caso seguiremos los mismos pasos y abordaremos la parte que concierne a los procesadores. Dentro de los procesadores realizaremos un estudio del funcionamiento dentro de sus memorias internas o cachés, buses y demás factores relacionados con la arquitectura de dichos procesadores.

Actualmente existen benchmarks de distintos tipos, tanto de código abierto (Linkpack <sup>[4]</sup>, NAS Pararell Benchmarks <sup>[5]</sup>), como de pago (SPEC <sup>[6]</sup>, Futuremark (3DMark) <sup>[7]</sup>). Los benchmarks se caracterizan por obligar al procesador a realizar un gran número de operaciones. Como se puede observar en la Figura 1, incluso un simple programa capaz de renderizar imágenes puede ser utilizado como benchmark. Esto se debe a que renderizar este tipo de imágenes puede llevar horas y horas de ejecución.



Figura 1: Imagen renderizada con 3DMark

Existen empresas que están dedicadas totalmente al desarrollo de este tipo de software. Los benchmarks siempre han generado tensiones en el mercado de los superordenadores, ya que son capaces de evaluar al más mínimo detalle, el comportamiento de cualquier tipo de computador actual. Es por ello que se va a implementar un software que intente ser neutral y que no aproveche las características de ninguna arquitectura en específico. Este tipo de neutralidad es difícil de alcanzar. A lo largo de la historia se han realizado benchmarks específicos para ordenadores, como por ejemplo el ICOMP <sup>[8]</sup>, creado por Intel para su línea de procesadores; y viceversa, ordenadores que se crean para pasar ciertos benchmarks con la mayor holgura posible.

La motivación general de este proyecto es la de crear un benchmark adaptativo y universal. Esto quiere decir que será capaz de adaptarse a la plataforma en la que se está ejecutando, de tal forma que, se consiga un mayor beneficio ejecutando este benchmark, con configuraciones que sean más óptimas o eficientes para dicha plataforma.

Además con benchmark universal queremos decir que este software se podrá ejecutar en cualquier tipo de máquina. Aunque es cierto que tendrá que tener un cierto software preinstalado, que se comentará en el Apartado 2.4.

A pesar de la complejidad de crear un benchmark, pretendemos que este código sea lo más transparente posible, y durante la siguiente memoria trataremos de explicar de forma clara y concisa los pasos que sigue el benchmark hasta alcanzar los resultados. Pretendemos por tanto, que la reutilización del todo o de ciertas partes del programa sea una tarea sencilla, así como que sea factible realizar ampliaciones y mejoras sobre la base del desarrollo.

## 1.2 Objetivos del proyecto

Uno de los principales objetivos de este proyecto es tener construido un benchmark capaz de ejecutar distintos tipos de funciones para evaluar el comportamiento del procesador. Además dentro de las distintas funciones pretendemos experimentar con distintos valores, como por ejemplo: el número de núcleos en los que se está ejecutando, o lanzar las tareas realizando distintos tipos de reparto de carga computacional, para los distintos núcleos que posea el procesador.

También se pretende ejecutar en al menos dos computadores distintos, de tal manera que se puedan evaluar los distintos resultados. Y realizar una comparativa sobre las distintas variables que procedamos a medir, en las respectivas máquinas. Siempre teniendo en cuenta que las variables a medir sean las mismas en ambos ordenadores. La comparativa se realizará mediante gráficas. Trataremos de analizar todos los resultados intentando acercarnos de la forma más precisa posible a lo que realmente está sucediendo la CPU.

Las pretensiones a la hora de implementar son las de utilizar software específico que sea utilizado de forma extendida, para que de esta manera podamos centrarnos directamente en el benchmark. Así no tendremos que perder el tiempo en leer y descifrar librerías, para luego tener que modificarlas. Gracias a los avances que se han producido en torno a los benchmarks, y a la creación de código abierto que soporta este tipo de aplicaciones, existen un gran número de herramientas que nos facilitarán la labor. En cuanto a este tipo de herramientas algunas de ellas han sido testeadas a lo largo de su historia y muchas de ellas se utilizan en universidades y centros de investigación. También tendremos en cuenta el soporte que puedan dar los creadores de ciertas herramientas, ya que, es probable que tomemos código y librerías de alguno de ellos, que quizás queramos modificar, para que el comportamiento se aproxime más a la idea de benchmark final que queremos realizar.

El desarrollo del proyecto se realizará en lenguaje C. Esto se debe a que la mayor parte de las librerías de ayuda que vamos a encontrar están en este lenguaje. Por otra parte, también es cierto que otro lenguaje bastante extendido en la creación de benchmarks es Fortran<sup>[9]</sup>. Éste siempre ha estado ligado a la computación de alto rendimiento. Sin embargo, algunas librerías específicas sobre funciones matemáticas no están disponibles en Fortran y es por ello que nos decantamos totalmente por C.

En la parte final del proyecto se pretenderá utilizar algún tipo de herramienta por la cual podamos conocer cuál es el entorno más beneficioso para el ordenador. En otras palabras, intentaremos conocer dentro de las distintas pruebas posibles, cuál es la configuración más óptima para la máquina. Para ello nos centraremos en alguna de las distintas variables de ejecución que poseerá el benchmark. Principalmente en el reparto de carga para los distintos núcleos dentro del procesador.

Por tanto, los objetivos son:

- Desarrollo de un benchmark de bajo nivel para analizar y comparar distintas arquitecturas.
- Extraer información del rendimiento del microprocesador mediante contadores hardware.
- Comparar equipos con rendimientos pico, es decir, ejecutar pruebas en situaciones poco comunes que beneficien a unas arquitecturas sobre otras.
- Evaluar y desarrollar técnicas de optimización para determinar la mejor configuración del benchmark en la que mejor se explota la arquitectura.
- Evaluar las dos plataformas reales con múltiples núcleos: Intel Core2Duo y AMD Opteron.



## 1.3 Introducción de la memoria

La memoria la podemos dividir en 6 partes principales:

- **Plataforma de evaluación:** en este apartado se tratan las dos arquitecturas en las que será lanzado el benchmark, describiendo las características técnicas de éstos. Se realiza una explicación de qué son los contadores hardware y cómo funcionan. Y hablamos del entorno software que utiliza la aplicación. En este entorno software tratamos los programas para ejecutar el benchmark en varios núcleos, la librería que utilizamos para la toma de datos y otros programas necesarios para la ejecución del benchmark.
- **Benchmarks de evaluación:** comentamos las diferentes funciones que han sido implementadas para evaluar el procesador. Además mostramos una pequeña aproximación de lo que debería suceder teóricamente antes de la parte de análisis y experimentación. Finalmente tratamos con técnicas de optimización para conseguir una configuración ideal de benchmark para una cierta CPU.
- **Experimentos:** es la parte donde se muestran los frutos de la implementación, esto es, los resultados de las diferentes funciones para distintas configuraciones. Intentaremos explicar de la forma más aproximada posible lo que realmente está sucediendo y el porqué de las gráficas resultantes.
- **Comparativa final:** resumiremos los resultados de las funciones para cada arquitectura y daremos una visión general de en qué contextos funciona mejor una arquitectura u otra.
- **Presupuesto:** donde tratamos de reflejar el coste aproximado del proyecto y la planificación de éste.
- **Conclusiones:** para finalizar en esta sección explicaremos alguna de las ideas finales del proyecto y de las dificultades y ventajas de adentrarse en un proyecto de este tipo.

Como el lector habrá podido comprobar en la parte inicial de la memoria posee un índice general, así como de un índice para las figuras y tablas que aparecen en la documentación.

Además en la parte final existen dos apartados: un apartado referencias, donde podrá comprobar las fuentes que fueron utilizadas durante la redacción de esta memoria y un apartado glosario donde podrá consultar las definiciones de los términos clave.

El documento también está formado por varios apéndices, que tratan de explicar con ejemplos prácticos, las librerías que fueron utilizadas para el desarrollo del benchmark.

Dentro de los apéndices se encuentra el manual de usuario donde comentamos los pasos que hay que seguir, así como las sentencias y parámetros a ejecutar para utilizar el benchmark.



## 2.-Plataforma de evaluación

El software será evaluado en dos plataformas. A primera vista, pueden parecer pocas, sin embargo la gran cantidad de datos que vamos a obtener de cada una de ellas y el amplio abanico de pruebas que pueden ser realizadas, hacen de esta elección una buena base para poder testear todos los objetivos iniciales del benchmark.

### 2.1 Intel Core 2 Duo T7100

Esta primera arquitectura está montada sobre un ordenador portátil Toshiba Satellite A200-1dy. Este portátil consta de dos placas de memoria RAM DDR2 de 512MB, una capacidad de 120GB, y el procesador es un Intel Core 2 Duo T7100<sup>[10]</sup>.

En esta máquina se desarrolla el código, aunque a priori sabíamos que la ejecución en el AMD, desencadenaría errores no controlados en el código inicial que implicarían cambios, debido a que son dos arquitecturas distintas.

La arquitectura del Core 2 Duo T7100 es la siguiente<sup>[11]</sup>:

Característica	Nº	Valor	Tipo	Más datos
Número de núcleos	2	1.800 GHz	-	-
Caché L1 Datos	2	32KB	Privada	Asociativa de 8 vías
Caché L1 Instrucciones	2	32KB	Privada	Asociativa de 8 vías
Caché L2	1	2MB	Compartida	Asociativa de 8 vías
Transferencia de bus	-	800 MHz	-	-

Tabla 1: Especificaciones técnicas de Intel Core de 2 Duo.

El tamaño de bloque de las memorias es de 64 Bytes, que es el tamaño con el que se van a comunicar entre sí las distintas memorias cachés. La política de reemplazos es LRU, es decir, el bloque de memoria que menos se utilice es que va a ser marcado para que cualquier bloque nuevo pueda ocupar su lugar. Para saber que significa asociativa de 8 vías vamos a explicar resumidamente cómo funciona la memoria internamente.

La memoria caché se encuentra integrada dentro del procesador y su implantación se debe a que es una memoria de rápido acceso. Al requerir un dato de la memoria principal, el procesador mirará antes si este dato se encuentra en alguna de sus memorias cachés. Aunque pueda parecer un paso más en algunos casos, está demostrado que la memorias cachés funcionan de forma eficiente, debido a que los programas suelen utilizar las mismas zonas de memoria (mismos datos) de forma redundante, por lo que los datos almacenados en las memorias cachés, son de crucial importancia a la hora de reducir tiempos de ejecución.

Cuando una zona de memoria no se encuentra precargada en la memoria principal, se produce lo que se conoce como un fallo en la memoria caché. En este caso se procede a cargar este bloque. La decisión de dónde se va a alojar viene dada por la función de correspondencia, si hablásemos de correspondencia directa estaríamos diciendo que cada bloque sólo puede ir

alojado en una zona de memoria concreta. Si hablásemos de una función de correspondencia asociativa de 2 vías, como es el caso de la Figura 2, que se muestra a continuación, estaríamos hablando de que cada bloque de memoria puede alojarse en dos zonas de la memoria caché.

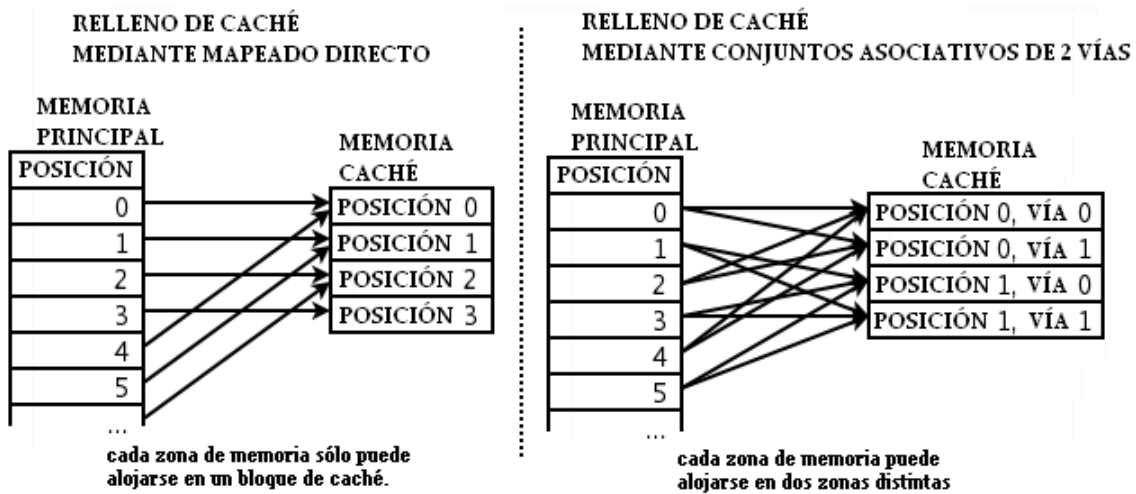


Figura 2: Funciones de correspondencia directa y asociativa por conjuntos de 2 vías.

En el caso del Core 2 Duo, cada bloque de memoria puede alojarse en 8 posiciones distintas, lo que puede parecer mucho. Sin embargo, cada bloque de memoria ocupa 64B, y teniendo en cuenta que la L1 de datos ocupa 32KB, estamos hablando de 512 posiciones distintas. La ventaja real está a la hora de buscar, ya que, sólo miraremos en 8 de las 512 posiciones para saber si tenemos un acierto o fallo caché. En el caso de L2 hablamos de más de 32 mil posiciones.

Una de las características a destacar de esta arquitectura es que utiliza la que Intel llama “Advanced Smart Caché”, que consiste en que la caché de nivel 2 es compartida. Comparada con una caché L2 privada para cada núcleo, la principal ventaja de este método es que ambos núcleos pueden compartir datos sin tener que acceder al bus de memoria. Por lo que reduce los accesos a memoria principal y optimiza el funcionamiento de L2 (desaparecen las redundancias).

A continuación se muestran ejemplos de la arquitectura del procesador Core 2 Duo <sup>[12]</sup>.

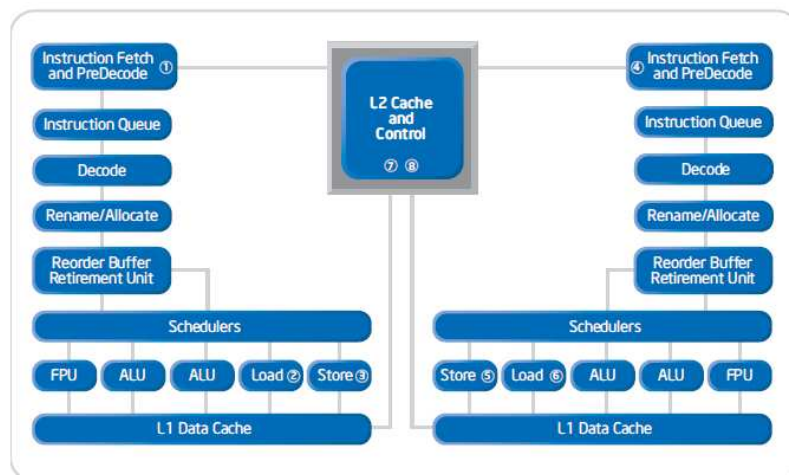


Figura 3: Arquitectura del procesador Intel Core 2 Duo.

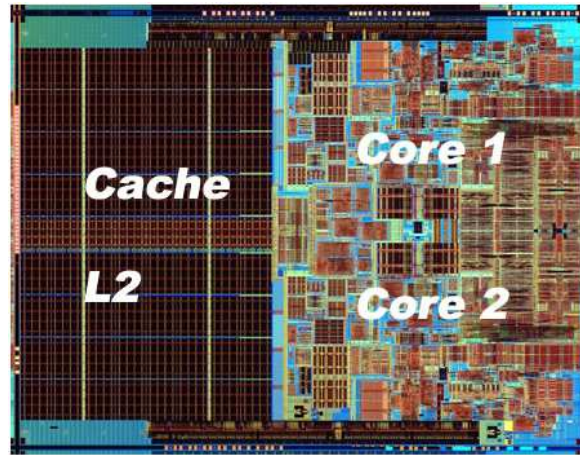


Figura 4: Gráfica de situación de la caché L2 en un Core 2 Duo

Además dentro de esta tecnología también se engloban técnicas de acceso inteligente a la memoria. La idea consiste en trabajar en los dos criterios más conocidos de acceso a memoria. Hablamos de las restricciones temporal y espacial. La restricción temporal dice que zonas de memoria que han sido accedidas recientemente volverán a ser accedidas en un futuro. La restricción espacial dice si una zona de memoria es accedida, es bastante probable que zonas de memoria contiguas sean accedidas.

Core 2 Duo introduce un mecanismo especulativo que predice si una lectura de una instrucción es susceptible de depender de escrituras que están siendo procesadas, de esta forma puede ser procesada sin tener que esperar a la finalización de las escrituras. Este rol predictivo acaba con las ambigüedades y se llama “Memory Disambiguation”. Detrás de esta reducción de la espera, el principal objetivo es reducir la dependencia entre las instrucciones incrementando la eficiencia general del programa<sup>[13]</sup>.

Esta arquitectura también posee *prefetching*. El *prefetching* es una técnica que intenta adivinar o predecir que instrucciones van a ser cargadas posteriormente. De tal forma, que cuando el bus se encuentra libre de trabajo, comienza a cargar en memoria caché, bloques de memoria que supuestamente van a ser utilizados. Sobre esta parte ampliaremos en el Apartado 2.4.3.1 y en la parte de la experimentación donde se habla de la desactivación de esta técnica.

## 2.2 AMD Opteron 6168

El segundo procesador que vamos a utilizar para la ejecución del benchmark va a ser el AMD Opteron 6168. La cesión de esta máquina para las pruebas fue gracias a la Universidad Carlos III y en concreto agradecemos su ayuda al profesor y doctor Francisco Javier García Blas, por configurar esta plataforma para que pudiese ser utilizada.

Mostramos las características principales <sup>[14]</sup>, al igual que hicimos con el anterior procesador:

Característica	Nº	Valor	Tipo	Más datos
Número de núcleos	12	1.900 MHz	-	-
Caché L1 Datos	12	64KB	Privada	Asociativa de 2 vías <sup>[15]</sup>
Caché L1 Instrucciones	12	64KB	Privada	Asociativa de 2 vías
Caché L2	12	512KB	Privada	Asociativa de 16 vías
Caché L3	2	6MB	Compartida	Se comparte en grupos de 6
Transferencia de bus	-	3200MHz	-	-

Tabla 2: Especificaciones técnicas de AMD Opteron.

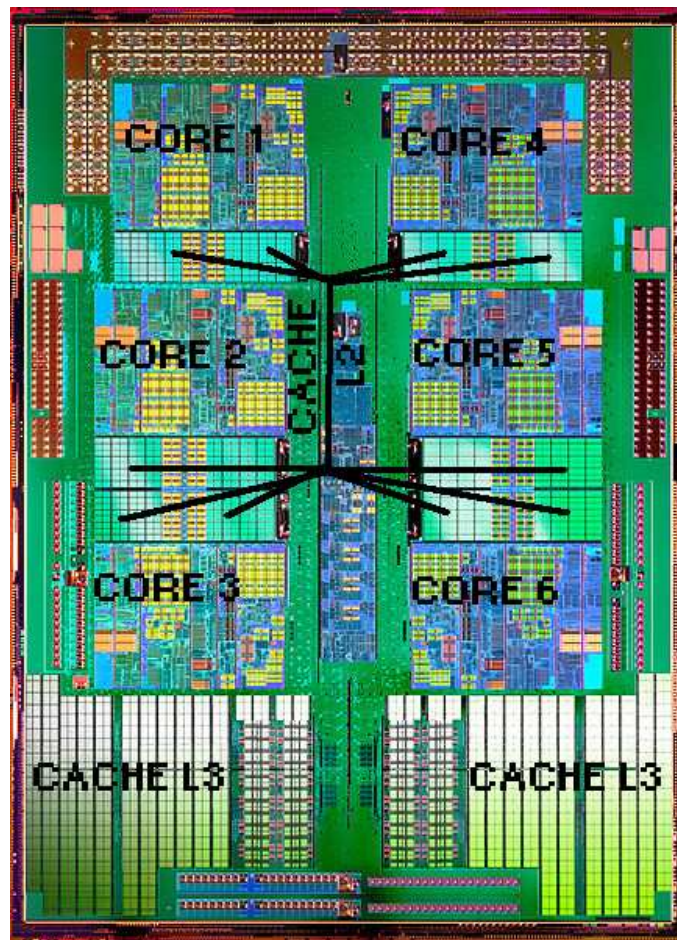
Si comparamos estos datos directamente con los del Core 2 Duo, las diferencias son abismales, así como su arquitectura interna. Para empezar el Opteron va a contar con un nivel más de caché y con 10 núcleos más.

Cada núcleo va a poseer su caché L1 privada de datos y de instrucciones, y una caché privada de nivel 2. En cuanto a la caché de nivel 3 se va a repartir, una para un grupo de seis núcleos y la otra para el otro grupo. De tal forma, que si tienen que compartir datos entre dos procesadores que no comparten L3, este dato deberá viajar por el bus.

En la Figura 5 veremos la disposición de los núcleos y de las memorias cachés en un AMD Opteron Istanbul con sólo 6 núcleos. Este microprocesador es de la misma familia que el Opteron Magny Course que utilizamos, pero con 6 núcleos menos. Cerca de cada uno de los núcleos tenemos las respectivas cachés de nivel 2 que, como hemos comentado antes son de tipo privado. Abajo del todo tenemos la memoria caché de nivel 3 bien diferenciada, que es compartida, al igual que sucedía en el nivel 2 de la anterior arquitectura descrita.

Dentro de cada uno de los núcleos en amarillo se visualizan las cachés de nivel 1, también diferenciadas en instrucciones y datos. Éstas también son de tipo privado.

Si nos preguntamos qué ventajas tiene que esta máquina posea tantos núcleos, tenemos que verlo desde el punto de vista de los programas que se ven beneficiados de ello. Por ejemplo, en el caso de que necesitemos ejecutar distintas máquinas virtuales vamos a tener un mayor rendimiento, también son buenos sistemas para bases de datos con constantes demandas, ya que puede resolver peticiones de forma paralela y, además, son muy ventajosos para los problemas que nosotros vamos a introducir, y con esto nos referimos al campo de la computación de alto rendimiento, más conocido como *high-performance computing* (HPC).



*Figura 5: Vista ampliada de un Opteron Istanbul*

Nuestra tarea va a ser desarrollar un problema que tengan un alto número de operaciones. Estas operaciones deberán ser preferiblemente independientes entre sí de tal forma que la carga se pueda dividir de forma independiente. Cada uno de los núcleos se va a encargar de realizar partes de la carga de trabajo, de tal forma que entre todos contribuyen a solucionar un problema global.

Otra de las ventajas con las que nos encontramos cuando trabajamos en este tipo de procesadores, es que cuentan con distintos canales de memoria. Los distintos canales permiten traer grandes bloques de datos para ser procesados y analizados <sup>[16]</sup>.

Si es cierto, que también tienen algún inconveniente y son los programas mono-hilo, o programas que no pueden ser paralelizados. En ocasiones sucede que algunos programas no pueden ser paralelizados debido al gran número de dependencias entre las operaciones y es por ello que, cuando se ejecutan en este tipo de procesadores, la ganancia no se corresponde de forma directa con el número de núcleos que posea. En estos casos sólo uno de los núcleos se encuentra trabajando mientras que los restantes permanecen ociosos.



## 2.3 Contadores hardware

Los contadores hardware son un conjunto de registros de propósito especial, que se incluyen dentro de los actuales microprocesadores. El objetivo de éstos es el de almacenar estadísticas o eventos que ocurren dentro del procesador. Estos contadores no afectan a la ejecución general del programa ya que, se encuentran implementados como hardware adicional.

En un principio puede parecer que no sean muy necesarios, pero si tenemos cuenta que estamos hablando en un contexto, en el que un supercomputador, como el Blue Gene/P realiza miles de millones de operaciones en un solo segundo y que este segundo de computación puede llegar a suponer un alto coste económico <sup>[17]</sup>; parece que tiene sentido adentrarse en el funcionamiento interno de éstos. Con este tipo de contadores, se intenta extraer el máximo rendimiento de este tipo de sistemas.

Frecuentemente el software se crea específicamente para cada tipo de ordenador en el que va a ser ejecutado. Los equipos de investigación estudian minuciosamente las estadísticas generadas por estos contadores, para medir el rendimiento exacto en sus aplicaciones y para descubrir configuraciones donde pueden conseguir mejoras adicionales.

Como hemos comentado anteriormente, la mayoría de las máquinas actuales se basan en el funcionamiento de carga de datos, mediante memorias cachés. Las memorias cachés ofrecen una mayor velocidad de acceso a los datos y a las instrucciones, de lo que es posible mediante la memoria principal. Por otra parte, la caché se diseña para cumplir los principios de localidad espacial y temporal. Dicho de otra forma, las cachés están diseñadas con la “esperanza” de que una aplicación vuelva a utilizar bloques de memoria que ya utilizó (temporal) y, a que acceda a bloques de memoria cercanos o contiguos a aquellos que ya fueron cargados previamente (espacial). Si una aplicación sigue estos principios, tendremos muchas más opciones de conseguir un alto rendimiento en un procesador cuyo funcionamiento se basa en el uso de memorias cachés. Si no, el rendimiento se desviará de las miras fijadas al comienzo del desarrollo de la aplicación.

El familiarizarse con este tipo de contadores toma algo de tiempo, pero es fácil conocer de forma rápida qué contadores están disponibles en nuestra máquina. Cada procesador tiene un número distinto de contadores, normalmente se nombran de manera distinta. Incluso dentro de diferentes familias o modelos los contadores que se encuentran disponibles pueden diferir enormemente. Sin embargo, como norma general los contadores se encargan de tomar medidas de eventos muy similares.

Dentro de este gran número de contadores podríamos realizar una somera clasificación, intentando agrupar estos en tres conjuntos:

- **Caché:** Como hemos explicado la mayoría de los sistemas actuales están basados en memorias cachés. El funcionamiento básico de un CPU, es el de intentar operar con datos que se encuentran en sus registros, si no están, deberá acceder a la memoria caché L1 y copiar esta información en sus registros; si esta información no se encuentra en L1, lo que se conoce como un falló caché en L1, deberá acceder a L2, y copiar el dato en L1 y en los registros; y así sucesivamente hasta encontrar

el dato. En el último nivel se encuentra el disco duro, pero esto sale del ámbito del procesador y como máximo estamos limitados al bus de comunicaciones entre la caché de más alto nivel y la memoria principal. A continuación mostramos un gráfico de las memorias (Figura 6), en cuanto a proximidad a la CPU. En algunos sistemas no se cuenta con memoria caché de nivel 3.

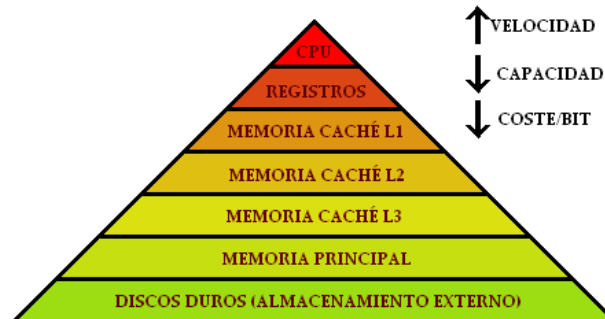


Figura 6: Pirámide de tipos de memoria en relación con CPU

Por tanto, vamos a tener datos de estadísticas de accesos, de aciertos (cuando el bloque accedido se encuentre en la memoria), fallos (cuando el bloque accedido no se encuentre en la memoria). La suma de aciertos y fallos es igual al número de accesos (intentos). Además también vamos a poder diferenciar entre los fallos/aciertos que se comentan tanto para instrucciones como para datos. En este caso con instrucción nos referimos a partes del programa que están siendo cargadas para ser ejecutadas y cuando nos referimos a datos, es la información que requiere el programa para seguir ejecutando. Dentro de todo este conjunto también vamos a poder diferenciar las operaciones de lectura y escritura que se realizan en las distintas memorias.

- **Instrucciones:** Otro tipo de contadores que podemos agrupar es el de las instrucciones. Dentro de un programa pueden generarse un gran número de instrucciones, sin embargo, las que resultan más costosas a un procesador, son las que conllevan operaciones en punto flotante. Estas operaciones implican sumas, multiplicaciones, divisiones y raíces cuadradas, de números de gran tamaño. También existen las denominadas instrucciones SIMD (*Single Instruction Multiple Data*) que pueden ser contabilizadas para precisión simple o doble. En cuanto a las instrucciones condicionales de salto también pueden ser contabilizadas teniendo en cuenta las veces que se dio o no el salto, y las veces que se predijo correctamente (o no) si se iba a dar el salto. Y en general también se pueden seguir las instrucciones que han sido emitidas y completadas.
- **Ciclos:** Dentro de esta división estaríamos agrupando los contadores que nos pueden dar información sobre número de ciclos que la máquina está parada, esperando por accesos de memoria, ya sean escrituras o lecturas. O también el número de ciclos que esperamos por cualquier otro recurso, como por ejemplo la unidad de coma flotante. También se puede contabilizar el número total de ciclos que se han llevado a cabo.

Después de esta visión general y clasificada, existen otros tipos de contadores que salen de este grupo y que se pueden mostrar ligados a una arquitectura, esto es, ser específicos de un microprocesador, o aparecer indistintamente en dos microprocesadores con arquitecturas muy dispares. Por ejemplo, el número de fallos en L3 no podrá aparecer en arquitecturas que no dispongan de este nivel de caché. Otro tipo de contadores que pueden aparecer son los relativos al bus, que conecta el microprocesador con la memoria, así como contadores relacionados con *prefetching*.

Los contadores además se clasifican en derivados y no derivados. Un contador derivado es un tipo de contador que se puede calcular a partir de otros contadores. Por ejemplo, el número de accesos (intentos) es igual al número de aciertos más el número de fallos.

Uno de los grandes problemas de los contadores es que el ordenador tiene un número limitado de registros y, por ello, no puede tomar estadísticas de todos los datos de una sola vez. Es aquí donde entra la multiplexación, mediante ésta se realiza la misma prueba para tomar medidas de distintos contadores. Es importante que la prueba suponga las mismas operaciones y el mismo resultado para el ordenador, puesto que de lo contrario estaríamos tomando medias erróneas. La multiplexación es necesaria a la hora de tomar estadísticas del ordenador, sin embargo supone un aumento importante en el tiempo de ejecución de los benchmarks.

La mejor fuente para conocer los eventos disponibles en nuestro procesador suele aparecer en la referencia técnica de nuestro procesador, frecuentemente disponible en la Web oficial del vendedor.

Otra complicación que podemos encontrar al nivel de *kernel* (núcleo del sistema operativo) es conseguir acceder a estos contadores hardware. Aunque cada vez más se va implantando en los sistemas operativos, e incluso el núcleo del Itanium (IA-64) provee del *driver* oficial (creados por Stephane Eranian<sup>[18]</sup>), el árbol estándar de versiones Linux para x86 no provee de esta característica.

Afortunadamente, se han realizado esfuerzos sobre estos temas. El primero y uno de los más importantes es el desarrollo de un *driver* de monitorización de estadísticas para el *kernel* x86 denominado *perfctr*. Éste es un parche muy estable desarrollado por Mikael Pettersson de la Universidad Uppsala en Suecia<sup>[19]</sup>. El parche *perfctr* cada vez está siendo más adoptado por la comunidad, y se mantiene y mejora de forma continua. El segundo esfuerzo proviene del Laboratorio de Computación e Innovación de la Universidad de Tennessee-Knoxville, y se denomina PAPI (Performance Application Programming Interface), traducido como Interfaz de Programación de Rendimiento de Aplicaciones. PAPI define un conjunto estándar de eventos que pueden ser monitorizados mediante una librería de sencilla instalación. El proyecto PAPI provee implementaciones para distintos sistemas operativos y para distintos tipos de procesadores. Esto incluye a los procesadores Intel/AMD x86, sistemas Itanium y más recientemente procesadores AMD x86-64. Expandiremos la información sobre PAPI en el Apartado 2.4.2 y en el Apéndice A.



## 2.4 Entorno software

Dentro de este apartado trataremos las distintas librerías y software que han sido utilizados para realizar el benchmark. Cabe destacar que todas ellas son “opensource” o de código abierto, por tanto cualquier puede tener acceso a ellas, así como a sus manuales y soporte online.

### 2.4.1 OpenMP

La API de OpenMP nos provee de computación paralela en distintas plataformas, con procesos capaces de compartir memoria <sup>[20]</sup>. Este código lo podemos utilizar tanto para programar en lenguaje C/C++ como para Fortran <sup>[21]</sup>, en todo tipo de arquitecturas, incluyendo plataformas Unix y Windows NT.

Dicho de forma más sencilla, mediante OpenMP vamos a conseguir paralelizar un simple programa, de tal forma que lo que antes se estaba ejecutando por un único proceso ahora se ejecutará en varios hilos. Para ello reparte la carga entre los distintos hilos, para que puedan trabajar conjuntamente y así terminar antes.

Con la paralelización vamos a explotar la ventaja que nos proporcionan los ordenadores con múltiples núcleos. Esto se debe a que cada hilo en el que se separa el programa principal, va a ser ejecutado en un núcleo distinto, y es en este punto donde se consigue realmente la paralelización del programa.

Concretamente, esta API nos va a permitir modificar ciertas variables dentro de cada prueba. Como hemos comentado vamos a poder tratar con distinto número de hilos, pero también vamos a poder experimentar con diferentes tipos de planificación y de reparto de carga. Entre estas variables se encuentran <sup>[22]</sup>:

- *Chunk*: o tamaño de grano, que determinará la carga que se va a asignar para cada hilo. Esta parte es importante, si elegimos mucha carga a repartir, puede suceder que no haya un buen balance entre procesos y que algunos realicen más trabajo que otros. Si elegimos una carga muy pequeña para cada proceso, tendremos también problemas puesto que no habrá reutilización de zonas de memoria y el *prefetching* no funcionará tan bien como en otros casos. Además en este último caso es probable que aumente el número de cambios de contexto.
- Nº de hilos o número de subprogramas en los que se dividirá el programa principal. Lo más correcto sería que fuese igual al número de núcleos del ordenador, ya que, si es mayor, habría procesos sin núcleos donde ejecutar. En caso contrario tendríamos núcleos en los que no se está ejecutando ningún hilo.
- Planificación: tipo de reparto que se realizará. OpenMP nos provee de las siguientes opciones: static, dynamic y guided, que veremos con más detalle en el Apartado 4.6.4.

Una de las ventajas más atractivas de esta librería, y quizás la que la ha hecho tan difundida, es que todos estos cambios se pueden incluir añadiendo simplemente dos líneas en

nuestro código original. Mediante ella tomaremos las medidas de tiempos gracias a la función `omp_get_wtime()` y será la única medida separada de tomadas con PAPI.

Como comentamos en el apéndice de PAPI y repetimos aquí, por ser de suma importancia, la toma de tiempos se hará de forma especial para eliminar el propio tiempo de ejecución de la rutina de tiempos. Para ello seguiremos los siguientes pasos: se toman tres medidas de tiempo, una antes del ejecutar el benchmark y dos después

Toma de tiempo T1

Ejecución de Benchmark

Toma de tiempo T2

Toma de tiempo T3

Y obtenemos el tiempo real de ejecución con la fórmula:

$$T_{\text{total}} = (T_2 - T_1) - (T_3 - T_2) = 2T_2 - T_1 - T_3$$

### 2.4.2 Librería PAPI

La librería PAPI será la que nos provea código para acceder a los contadores hardware<sup>[23]</sup> de los que hemos hablado al comienzo del Apartado 2.4. De esta forma vamos a obtener la información de lo que está sucediendo, en cuanto a rendimiento, después de la ejecución del benchmark. En concreto utilizaremos la versión 4.0.0.

El uso de esta librería está bastante extendido y cuenta con varias actualizaciones a sus espaldas, por lo que es una librería bastante robusta. Además posee un manual bastante detallado con ejemplos, que son una guía perfecta para arrancar con la herramienta<sup>[24]</sup>.

Como soporte posee un foro bastante activo, con usuarios que pueden ayudar en caso de dudas, o para resolver algún tema que quede poco claro en el manual<sup>[25]</sup>.

Dentro de la librería PAPI se hace una clara diferencia entre los eventos preestablecidos y los eventos nativos. Los eventos preestablecidos son un conjunto de eventos que se pueden encontrar típicamente en muchas CPUs. Por otra parte, los eventos nativos son eventos propios de una arquitectura en concreto y, por tanto, no pueden ser utilizados en todas las plataformas existentes.

Para utilizar PAPI podemos hacerlo utilizando cualquiera de sus dos interfaces, de alto y bajo nivel. Ambas proveen de funciones para arrancar, parar y leer los contadores de una lista específica de eventos, aunque las diferencias son constatables. Por su parte, la interfaz de alto nivel consigue, de forma más sencilla para el programador, dar acceso a todo tipo de información. Sin embargo, si deseamos tener un mayor control de la aplicación, así como poder acceder indistintamente a eventos preestablecidos como eventos nativos; deberemos hacer uso de la interfaz de bajo nivel. Quizás esta última razón premie sobre todas y ha sido la que nos ha llevado a utilizar la interfaz de bajo nivel de PAPI. Ya que, en nuestro interés está el de usar conjuntamente eventos nativos (específicos de una cierta arquitectura) y eventos preestablecidos.

PAPI también provee de multiplexación. La multiplexación permite leer más contadores de los permitidos por los registros de la plataforma. De esta forma, se le pasa una lista de

contadores y PAPI se encarga de tomar las estadísticas de todos ellos. Uno de los grandes problemas de la multiplexación es que no funciona en todas las plataformas y en concreto en la plataforma en la que vamos a implementar el código, Intel Core2 Duo, no funciona, por lo que en nuestro caso hemos optado por implementar la multiplexación manualmente. Para ello debemos pasar un conjunto limitado de eventos y después realizar las pruebas de nuevo tantas veces como sea necesario hasta que acabemos con el resto de contadores que deseemos monitorizar. Una descripción detallada de los pasos que seguiremos viene en el Apartado 4.1.

La versión que vamos a utilizar de PAPI (4.0.0) no cumple con las normas de Valgrind. Valgrind es un software capaz de detectar fugas de memoria y que es útil para este tipo de software <sup>[26]</sup>, así nos aseguraríamos de que toda la memoria que se utiliza se libera al final de la ejecución, evitando provocar que toda la memoria reservada no sea liberada. Debido a este problema no podremos asegurarnos de que nuestro código no pierda memoria ya que se incrusta junto con las funciones de PAPI y no podemos analizarlo de forma independiente.

En el Apéndice A, se muestra un ejemplo de uso de las librerías de PAPI, en una interfaz de alto y de bajo nivel.

### 2.4.3 Prefetching

El *prefetching* es una técnica bastante extendida en los microprocesadores actuales. Ésta consiste en cargar bloques de memoria en la memoria caché, ya sean datos o instrucciones, que se supone que van a ser utilizados, ya sea por proximidad o por probabilidad. Las técnicas de *prefetching* intentan predecir de forma más o menos acertada qué zonas de memoria serán utilizadas, para ello realizan un estudio del patrón de acceso a memoria que realiza la CPU en la memoria, y si éste se aleja de la aleatoriedad, el prefetching va a proporcionar muy buenos resultados <sup>[27]</sup>, dado que va a predecir y cargar en la caché los datos que van a ser accedidos en el futuro.

Dentro del *prefetching* se pueden distinguir dos tipos, *prefetching* software y hardware. El *prefetching* software lo realiza el compilador. Éste añade instrucciones de precarga al código original en zonas donde piensa que van a utilizarse datos o instrucciones. Por su parte el *prefetching* hardware funciona totalmente desligado al código y, de forma independiente, carga datos o instrucciones que tienen alta probabilidad de ser utilizados.

La idea es realizar un estudio con las cuatro posibilidades existentes:

Prefetching	Hardware	Software
Experimento A	✓	✓
Experimento B	✓	-
Experimento C	-	✓
Experimento D	-	-

Tabla 3: Pruebas a lanzar con distintas configuraciones de prefetching

Es decir, activaremos y desactivaremos los distintos tipos de *prefetching* y veremos su comportamiento. Esta idea no es nuestra original y la hemos sacado del siguiente paper de

Intel: *Optimizing Embedded System Performance — Impact of Data Prefetching on a Medical Imaging Application* [28]. En el que entre otras cosas, se utiliza un generador de imágenes médico (AMIDE [29]). Y como medida se toma el tiempo que tarda el procesador en renderizar una imagen. En la Figura 7 mostramos una captura del AMIDE (A Medical Imaging Data Examiner) software que sirve para analizar y registrar imágenes médicas volumétricas. AMIDE además es un software de licencia libre.

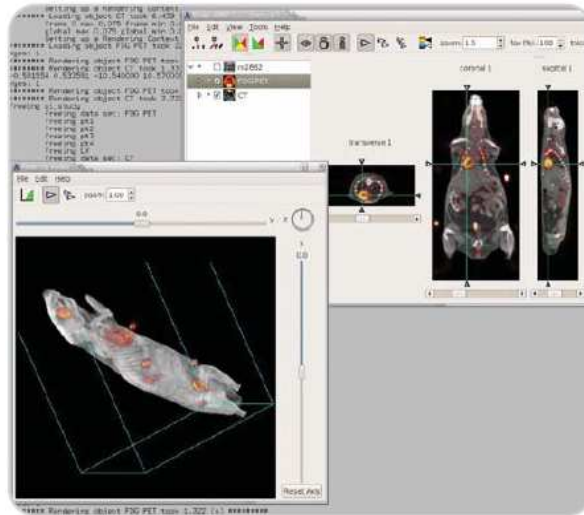


Figura 7: Captura del renderizador de imágenes médicas AMIDE.

#### 2.4.3.1 Prefetching Hardware, MSR-TOOLS

El hardware prefetch no es una nueva técnica. Comenzó a ser utilizada con el Pentium III Tualatin. Sin embargo, fue el Intel Netburst quien lo mejoró fundamentalmente. La gran diferencia entre la frecuencia del procesador y del bus, hizo al Netburst particularmente sensible a los efectos de los fallos cachés, y aumentó fuertemente el interés por un *prefetching* eficiente.

La herramienta MSR-Tools es un software que nos permite acceder a los registros del ordenador, tanto en modo lectura como en modo escritura [30]. Uno de los grandes inconvenientes de este software es la falta de manual y de soporte.

Con este software desactivaremos las opciones de *prefetching* del Intel Core2Duo y realizaremos un estudio de lo que está sucediendo.

Mediante esta herramienta MSR-TOOLS, vamos a ser capaces de desactivar el *prefetching* hardware. Si bien es cierto que en algunos ordenadores esta opción se puede deshabilitar en la BIOS (véase Figura 8), en nuestro caso no disponemos de esa opción y por eso requerimos de esta herramienta.

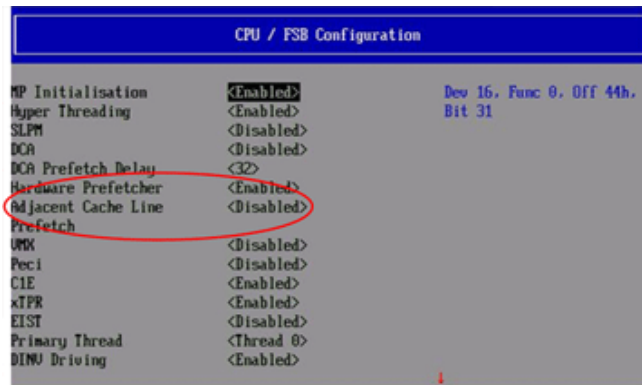


Figura 8: Captura en la que la BIOS permite desactivar el prefetching hardware

Para saber que registros vamos a utilizar deberemos consultar la documentación de Intel <sup>[31]</sup> y de AMD <sup>[32]</sup>, para acceder a los respectivos registros que modifican estas configuraciones.

Uno de los grandes problemas de esta herramienta es que tiene un control de errores muy pobre, ya que podemos introducir cualquier nombre de registro y en la mayoría de los casos no retorna un fallo, diciendo que este registro no existe.

Por ejemplo, si deseamos leer un registro usaremos:

```
rdmsr -x 0x1A0 _____ registro
                        _____ tipo salida (hexadecimal)
                        _____ comando
1364972489 (salida hexadecimal)
```

Y para escribir en el registro de un procesador en concreto:

```
wrmsr -p0 0x1A0 0xB3649F2689 _____ valor
                        _____ registro
                        _____ núcleo (p0,p1...)
                        _____ comando
```

El caso es que si hacemos un rdmsr del registro “registroinexistente” nos devuelve valores, como si hubiese un registro con ese nombre. Después de probar distintas combinaciones con el registro “MSRC001\_1022 Data Cache Configuration Register (DC\_CFG)” fue imposible modificar los registros en el Opteron. Además cada vez que realizábamos intentos, lanzábamos el benchmark pero las ejecuciones arrojaban los mismos resultados, por lo que no ha sido posible desactivar el hardware *prefetching* del AMD.

Dentro del Intel Core 2 Duo podemos hablar de distintos mecanismos de *prefetching*:

- El prefetcher de instrucciones precarga instrucciones en la caché de instrucciones L1, basándose en predicciones de salto. Cada uno de los dos cores tiene uno.
- El IP prefetcher (*Instruction Pointer-based prefetcher* o el prefetcher de instrucciones basado en punteros) escrudiña el histórico de lecturas para tener un

diagrama generalizado y cargar “futuros bloques útiles”, de datos, en la caché de datos de nivel 1. Cada uno de los dos cores tiene uno.

- El DCU prefetcher (*Data Cache Unit* – Unidad de datos caché) detecta múltiples lecturas de una simple línea de caché durante un determinado tiempo y decide cargar la siguiente línea en la caché L1. También una por cada core
- El DPL prefetcher (*Data Prefetch Logic* – Carga de datos lógica) tiene un comportamiento similar al DCU Prefetcher. La única diferencia es que detecta peticiones en dos líneas consecutivas de caché (N y N+1) y se dispara si la lectura de la línea N+2 mueve el bloque de la memoria a la caché L2. La caché L2 también tiene dos de ellas, pero se comparten dinámicamente entre ambos núcleos.

El número total de prefetchers que hay en el Core2Duo son por tanto los ocho que hemos tratado y se representan mediante soles en la siguiente figura:

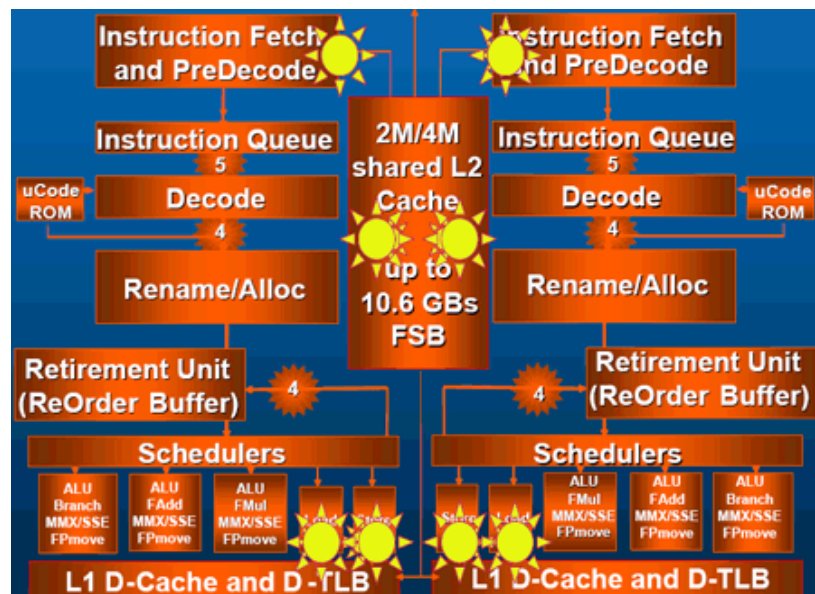


Figura 9: Prefetchers dentro del Intel Core2Duo

En la Figura 9 vemos cómo arriba del todo tenemos los prefetchers de instrucciones, en la mitad tenemos los dos DPL prefetchers que se comparten en L2. Y abajo del todo localizamos el IP prefetcher y el DCU, privados y uno para cada núcleo.

Veamos ahora a bajo nivel cómo desactivar el *prefetching*. Intel nos dice que dentro del registro 0x1A0, tenemos los siguientes bits:

- 39 (Desactivar IP Prefetcher)
- 37 (Desactivar DCU Prefetcher)
- 19 (Desactivar Adjacent cacheline prefetch)
- 9 (Desactivar Hardware Prefetcher)

Vamos a ver en la Figura 10 cómo hay que cambiar los valores para que queden modificados realmente:

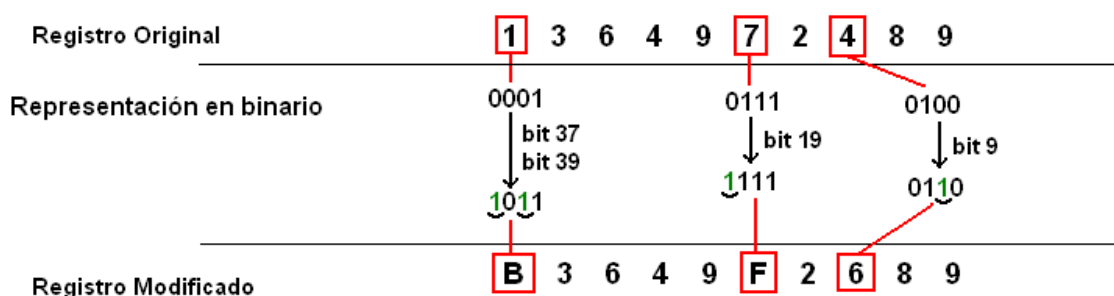


Figura 10: Prefetchers dentro del Intel Core 2 Duo

De esta forma modificamos el registro para desactivar todas las opciones de *prefetching*. Cabe destacar que al reiniciar el ordenador los registros se resetean y vuelven a tener los valores de origen.

#### 2.4.3.2 Prefetching Software

Para activar y desactivar el software *prefetching* lo tenemos mucho más fácil. En nuestro caso estamos utilizando el compilador gcc<sup>[33]</sup>. Este compilador provee de distintas opciones de optimización, para conseguir que nuestro código sea más eficiente<sup>[34]</sup>. Una de ellas es O3, que utilizaremos durante todas las pruebas, sin embargo, existe una opción de optimización que desactiva el flag de software *prefetching*, esta es Os.

Mediante esta opción conseguiremos que el compilador deje de añadir nuevas instrucciones al código, de posibles líneas de código que van a ser cargadas. Al desactivar esta opción veremos qué efectos tiene sobre el resultado final del benchmark, y para ello realizaremos un estudio de lo que sucede con los distintos contadores utilizados.

#### 2.4.4 Huge Pages

Huge Pages no se puede considerar un software, como en los tres apartados anteriores que hemos visto, si no como un conjunto de pasos o procedimientos para aumentar el número de páginas de las que va a disponer un proceso.

La TLB o Translation lookaside buffer, es una pequeña memoria que la Unidad de gestión de memoria (MMU – Memory Management Unit), utilizada para aumentar la velocidad de traducción de la unidad virtual. El primer paso que realiza el microprocesador cuando no posee un dato en sus registros, antes de mirar en la memoria caché, es buscar en la TLB la página donde se encuentra guardado el dato/instrucción. De esta forma, en caso de encontrarse esa información en la TLB se denomina acierto TLB y si no, fallo TLB.

En caso de acierto TLB, se toma la página donde está guardada la información requerida y se traduce a direcciones físicas donde accederemos a ellas de forma directa. En caso de fallo la MMU generará una excepción de fallo de página, que el sistema operativo se encargará de manejar. Para solucionar esta excepción el sistema operativo deberá cargar los datos solicitados en la memoria y actualizar la tabla de páginas.

En algunos casos un benchmark puede requerir un gran número de datos, y estos probablemente no quepan en la memoria caché. Además estos datos no van a estar ordenados por orden de acceso, que sería lo ideal. Sin embargo, con esta técnica vamos a conseguir aumentar el número de páginas por proceso, con lo que vamos a conseguir un mayor aumento de la tasa de aciertos TLB. Además conseguiremos mapear la memoria mediante una función, para que se reserve de forma consecutiva, y así, nos aseguraremos de que la memoria caché se encuentra mapeada de la mejor forma posible.

El tamaño medio por página es de 4096 Bytes en la mayoría de arquitecturas x86. En las dos arquitecturas que vamos a testear se cumple esta premisa.

Por lo tanto, los pasos que vamos a seguir son:

- Modificar el número de páginas a reservar, para ello ejecutaremos el comando `echo 4 > /proc/sys/vm/nr_hugpages` (estaríamos reservando cuatro páginas), si cada página ocupa un total de 4096 Bytes, estamos hablando de 16KB.
- Comprobar que la modificación se ha realizado con éxito, en ocasiones el anterior paso puede llevar cierto tiempo. La comprobación se hace con:  
`cat /proc/meminfo`
- Compilar y ejecutar el código

Antes sin embargo, debemos asegurarnos que la reserva de memoria se hace sobre las páginas de memoria que hemos reservado. Si reservamos más memoria de la que poseemos, tendremos un error “shmget: Invalid argument”.

Una aplicación puede hacer uso de las huge pages de dos formas. Una es utilizando una región especial de memoria compartida y la otra es mediante la función `mmap`, que mapea archivos desde el sistema de archivos de la TLB. Especialmente si deseamos utilizar mapeo de Huge Pages de forma privada, se recomienda utilizar `mmap`. En nuestro caso seguimos un tutorial en el que conseguimos utilizar estas páginas mediante memoria compartida.

En ambos casos deberemos sustituir la función `malloc`, y las dependencias de esta, por las funciones `shmget` y `shmat`. En el Apéndice B, dejaremos explicado el código que se debe incrustar en la parte original, así como el proceso global. En el Apartado 4.7 de experimentación veremos los distintos resultados arrojados.



## 3.- Benchmarks de evaluación

---

Hasta ahora ya hemos visto todas las herramientas en las que nos vamos a apoyar para realizar el benchmark. Sin embargo, no hemos tratado en qué va a consistir éste, o qué pruebas vamos a realizar.

Para conocer el funcionamiento de un microprocesador básicamente tenemos que fijarnos en la parte fundamental de éste, que es la comunicación con sus respectivas memorias cachés, y el modo de uso que hace de ellas. Es por ello, que la mayoría de benchmarks tratan de recorrer zonas de memoria de distintas formas: secuencialmente, mediante saltos o de forma aleatoria <sup>[35]</sup>; tratando así de evaluar el funcionamiento interno del procesador.

En este apartado abordaremos los distintos planteamientos surgidos para testear ambos microprocesadores y trataremos de predecir los problemas con los que se encontrará en las distintas situaciones.

### 3.1 Escritura con stride

Esta prueba consiste simplemente en recorrer una matriz, e ir escribiendo en ella. De aquí derivan dos posibilidades, recorrer esta matriz por filas, es decir, siguiendo el orden normal por el que están almacenadas en memoria o, recorrerla por columnas. Cuando accedemos por columnas nos referimos obviamente a acceder primero al elemento que se encuentra en la primera fila y primera columna, luego al elemento de la segunda fila y primera columna y así sucesivamente. Esta prueba se ejecuta para un único hilo.

A continuación mostramos un ejemplo gráfico del acceso a la matriz por filas y columnas:

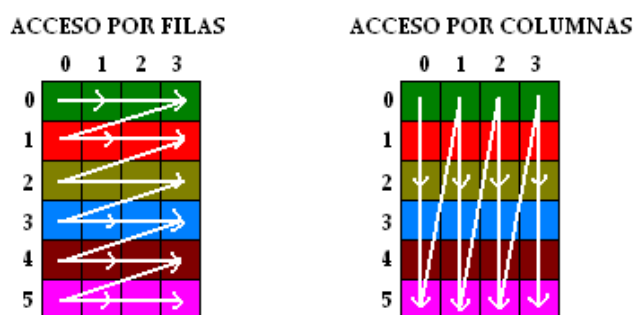


Figura 11: Acceso a matrices por filas y por columnas

Aunque la implementación de los códigos es muy similar, existe una diferencia abismal desde el punto de vista acceso a la memoria caché. Ya que en el primer caso estamos cargando bloques de memoria que se encuentran almacenados de forma consecutiva. Debemos ver la matriz como un simple vector. Sin embargo, en el acceso por columnas estamos accediendo mediante saltos, de aquí el título de esta prueba “*STRIDE*”.

Al acceder con estos saltos afectamos al rendimiento de la caché de forma importante, ya que para cada elemento que va a cargar, tendrá que tomar un mayor número de bloques, en algunos casos requerirá de toda la fila. Desde el punto de vista de la memoria cuanto mayor

sea el tamaño de la fila (o mayor sea el número de columnas); menos localidad espacial podremos encontrar, ya que los elementos que vamos a acceder de forma iterativa se van a encontrar más distanciados. La separación entre los elementos viene dada por la siguiente expresión:

$$\text{Distancia ( elto N , elto N+1 )} = \text{tamaño de la fila (número de columnas)}$$

Algo similar sucede en el caso del número de filas, cuanto mayor sea el número de filas también aumentarán los problemas. El caso ideal sería que tuviésemos cargadas todas las filas en memoria sin que el procesador tuviese que eliminar ningún bloque de la caché. Y que cuando el procesador iterase por segunda vez, tuviese cargado en memoria todas las filas de la matriz. En este caso estamos hablando de localidad temporal. Para conseguir que todas las filas se encuentren en memoria debería cumplirse lo siguiente:

$$\text{Tamaño de la caché} \geq (\text{Tamaño de bloque caché} * \text{número de filas})$$

En el caso de ser iguales sólo tendríamos un fallo la primera vez que cargásemos un nuevo bloque, pero podríamos reutilizar el bloque tantas veces como elementos quepan dentro de él.

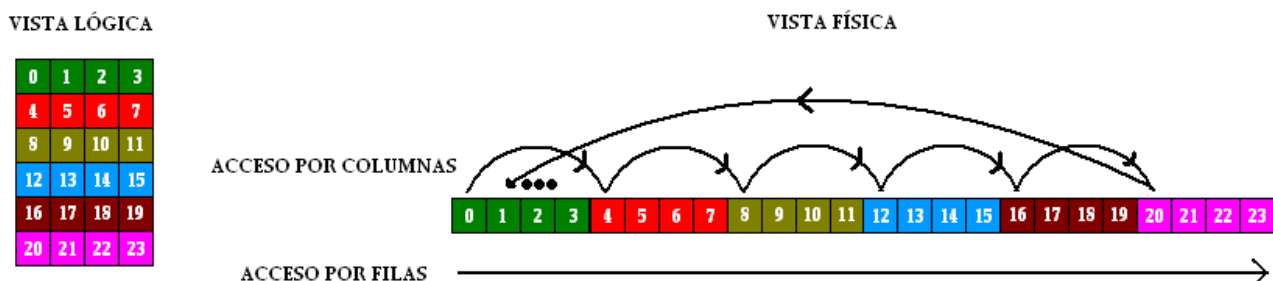
Sin embargo, en nuestro caso y para asegurar la equidad de las pruebas, nos aseguraremos de que el número de elementos de la matriz siempre sea el mismo y, por tanto, al aumentar el número de filas de la matriz disminuiríamos el número de columnas de ésta.

$$\text{Número de columnas} = \text{Número de elementos} / \text{número de filas}$$

Por tanto, según aumentemos el número de filas, el salto será menor pudiendo llegar a ser tan pequeño que podría estar cargado en el mismo bloque de memoria.

Otro detalle importante es que si deseamos que el número de elementos se mantenga constante variando el número de filas, el número de filas será siempre múltiplo del número de elementos totales de la matriz, así todas las filas ocuparán lo mismo.

Vamos a mostrar una vista física de lo que ocurre con ambos accesos, a nivel de memoria.



Es evidente que un acceso por columnas con tamaño de fila igual a uno, o con tamaño de columna igual a uno, va a ser igual que un acceso por filas. El estudio de este caso es interesante desde el punto de vista de *prefetching*. Se prevé que al aumentar el número de filas, manteniendo el mismo número de elementos, habrá un mayor número de fallos en las memorias cachés, y los tiempos de ejecución sean mucho mayores.

El texto a continuación muestra el código:

```
for(rep=0; rep < 1000; rep++){
    for(i=0; i < n1; i++){
        for(j=0; j < n2; j++){
            A[(j*n1)+i] +=rep;
        }
    }
}
```

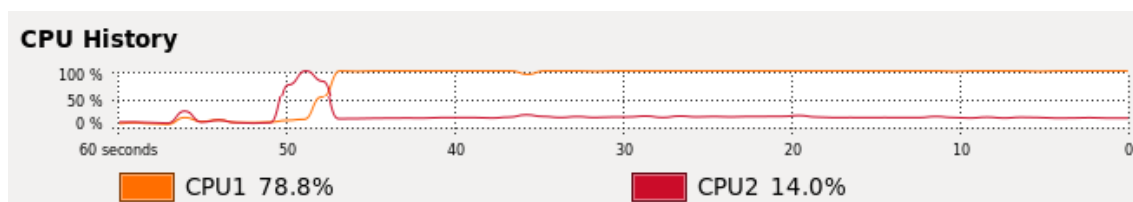
*Código 1: Escritura con stride*

Se realizan varias repeticiones para aumentar la carga de trabajo y vamos acumulando dentro de la matriz el valor de la variable rep.

Al finalizar la ejecución de esta prueba se muestra la suma de todos los valores de esta matriz. De esta forma nos aseguramos de que el compilador no elimine partes de código que podrían no ser necesarias para la ejecución y, además podemos comprobar que el resultado sea siempre el mismo. En caso de que los resultados fuesen distintos no estaríamos realizando un buen benchmark, ya que, para la multiplexación es importante que la prueba sea siempre la misma, así sabremos a ciencia cierta que las lecturas de los contadores que estamos realizando son correctas.

Para esta versión de escritura con stride también existen pruebas a ejecutar con modo sólo lectura, en las cuales la matriz se inicializa al comienzo de la ejecución y se recorre, por filas o columnas, acumulando los valores de esta matriz en una variable que se imprime al final.

En la siguiente figura mostramos la salida del monitor del sistema, que nos dice la ocupación de ambos núcleos en el Core2Duo durante la ejecución de esta prueba. Como podemos ver en este caso sólo está trabajando un único núcleo, en este caso el número 1:



*Figura 13: Ocupación de los núcleos en la prueba escritura con stride*

### 3.2 False sharing

El código de *false sharing* almacena valores en un simple vector. En este caso entra en juego la librería de OpenMP, esto se debe a que *false sharing* será ejecutado por dos o más hilos.

*False sharing* proviene del inglés, y hace referencia al hecho de que dos procesos simultáneos que comparten zonas de memoria, en el caso de que uno de ellos modifique esta zona de memoria, provocará que la información que cargó el otro proceso sea una información errónea, falsa, desactualizada.

Para realizar las distintas pruebas, jugaremos con el tamaño de grano. El tamaño de grano se puede explicar como la carga que se reparte para cada proceso, en nuestro caso concreto un tamaño de grano 8, indicará que cada proceso realizará 8 operaciones de almacenado en el vector.

Mostramos en la siguiente figura un pequeño ejemplo del reparto de tareas en *false sharing* para distintos tamaños de grano con dos procesos.

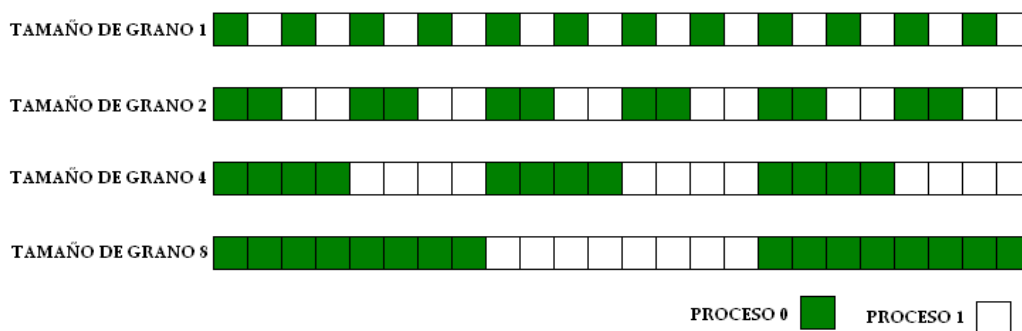


Figura 14: Ejemplo de reparto de tareas en *false sharing*

En la Figura 14 se puede apreciar claramente que cuanto menor sea el tamaño de grano, mayor zona de memoria van a compartir los procesos. El peor de todos los casos es tamaño de grano 1, en el que ambos procesos van a compartir todas las zonas de memoria sobre las que vamos a trabajar. Por otra parte, el caso ideal sería aquel en el que el tamaño de grano fuese igual al tamaño de bloque de la memoria caché, de esta forma las zonas serían independientes entre sí, y cada proceso podría trabajar sobre porciones de memoria independiente.

Para conocer el funcionamiento interno de esta prueba es importante, tener en cuenta el protocolo MESI, que nos aclara el funcionamiento interno de los bloques dentro de la memoria. La arquitectura Core2Duo utiliza MESI <sup>[36]</sup>, mientras que la del Opteron utiliza una variación llamada MOESI <sup>[37]</sup>. Según MESI cada línea de memoria caché puede estar en uno de los cuatro estados siguientes:

- Modificado: la línea caché se encuentra modificada, por lo cual tendrá que ser actualizada en memoria principal para que tenga el valor correcto.
- Exclusivo: la línea caché se encuentra en la caché actual, y su valor coincide con el de la memoria principal.
- Compartida: la línea puede estar duplicada en las distintas cachés de los distintos núcleos.

- **Inválida:** indica que la línea de caché ha sido modificada por otro proceso, y debe ser cargada de nuevo.

MOESI añadiría un único estado más O (*Owner*) que representa el estado en el que el bloque está modificado y compartido al mismo tiempo.

Pero para saber cómo funciona realmente MESI, vamos a realizar una pequeña simulación teórica e hipotética de ejemplo, sobre lo que podría suceder. Para ello vamos a hacer una pequeña traza de lo que ocurriría a nivel de caché, con un tamaño de grano igual a uno y trabajando con dos procesos. En el caso de los benchmarks que vamos a realizar el tamaño de la matriz siempre va a superar el tamaño de la memoria caché de nivel 2, pero para hacerlo más sencillo vamos a suponer una única caché con dos bloques y suponemos que la matriz ocuparía cuatro bloques.

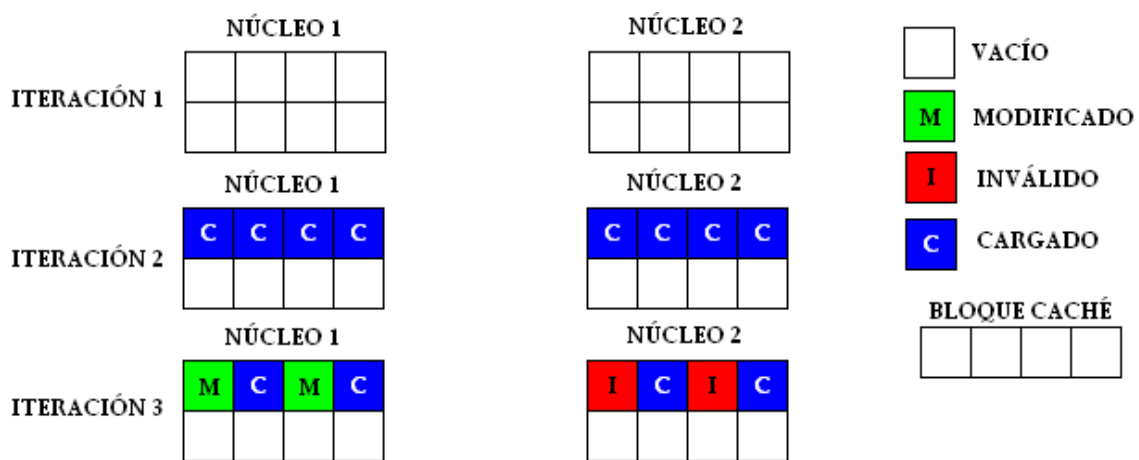


Figura 15: Ejemplo1 traza false sharing

En este ejemplo de la Figura 15 ambas cachés comienzan vacías y cargan el primer bloque caché, por lo tanto aquí tenemos los dos primeros fallos en memoria caché, en el siguiente paso el núcleo 1 escribe sus valores 0 y 2, y por tanto invalida los valores que poseía el núcleo 2, que tendrá que volver a cargar ese bloque (+1 fallo caché).

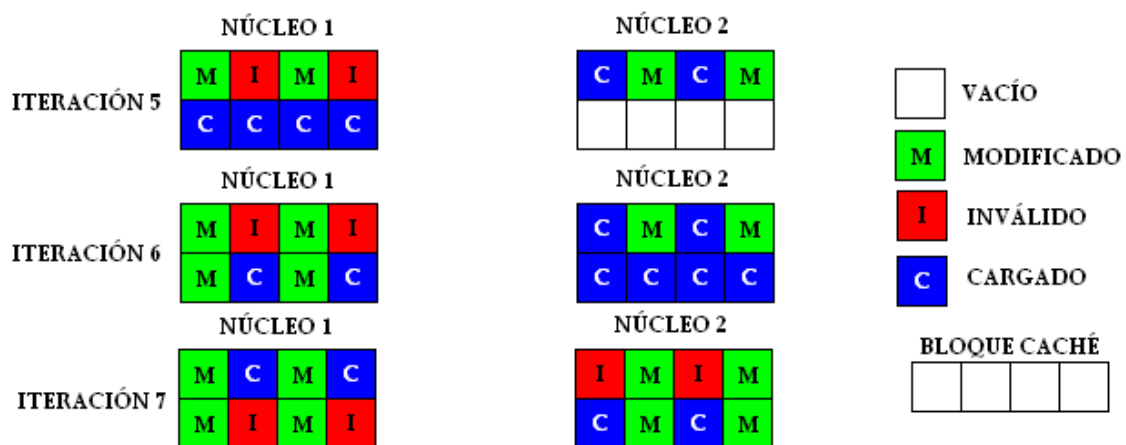


Figura 16: Ejemplo2 traza false sharing

Vamos a suponer que en la iteración 5 (Figura 16) el núcleo uno carga el siguiente bloque de memoria, pero mientras tanto el núcleo 2 escribe en su bloque 1, por lo que vuelve a invalidar el primer bloque que cargó el núcleo 1. Después de esto, en la sexta iteración, el núcleo 1 modifica los datos del segundo bloque, y suponemos que posteriormente el núcleo 2 carga todo modificado correctamente. En la última iteración que vemos, el núcleo 2 escribe en el bloque 2, por lo que vuelve a invalidar y lo mismo sucede con el núcleo 1, que después de haber cargado escribe en el bloque 1.

Lo que queremos mostrar con este ejemplo es que ambos núcleos van a estar constantemente invalidando sus respectivas cachés y, por tanto, van a tener que acceder de nuevo a la memoria principal (o a una superior memoria caché compartida) para recuperar los datos actualizados. Como hemos dicho, este ejemplo es uno de los peores casos posibles y en una situación ideal si el tamaño del bloque fuese igual al tamaño de grano, cada núcleo poseería su propia zona de memoria sobre la que trabajar sin estar continuamente interfiriendo uno sobre otro.

En el siguiente cuadro vemos la parte principal del código:

```
omp_set_num_threads(num_threads);
for(rep=0; rep < 1000; rep++){
    #pragma omp parallel for shared(A) schedule(SCHEDULE,chunk)
    for(j=0; j < n1; j++){
        A[j] +=rep;
    }
}
```

Código 2: False sharing

En la primera línea estamos estableciendo el número de hilos que ejecutarán el *parallel for*. Al igual que en escritura con stride tenemos varias repeticiones del mismo código, y como diferencia tenemos la línea de *pragma*, que es la que nos permite paralelizar el programa. En este caso contamos con un solo *for*, ya que sólo estamos recorriendo un vector.

En la siguiente imagen vemos cómo en el monitor del sistema ambos núcleos (Core2Duo) funcionan al 100%:

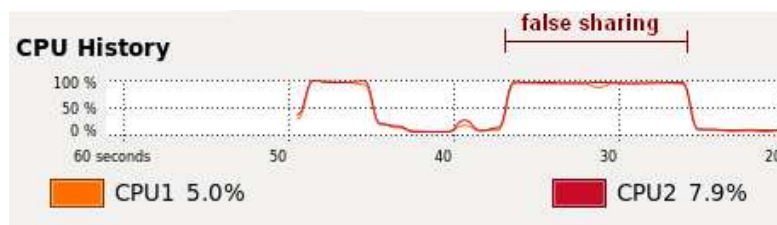


Figura 17: Ocupación de los núcleos en la prueba false sharing

### 3.3 Producto matriz-dispersa vector

El producto matriz-dispersa vector consiste, como su propio nombre indica, en ejecutar un producto entre una matriz dispersa y un vector. Cuando realizamos el producto entre una matriz normal y un vector, simplemente multiplicamos las filas de la matriz por el vector y de esta forma obtenemos el producto, que es a su vez un vector.

$$\begin{array}{ccc} \mathbf{A} & \mathbf{B} & \\ \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} & \cdot \begin{pmatrix} 5 \\ 6 \end{pmatrix} & = \begin{pmatrix} 5+12 \\ 15+24 \end{pmatrix} = \begin{pmatrix} 17 \\ 39 \end{pmatrix} \\ \mathbf{m \times n} & \mathbf{n \times o} & \mathbf{m \times o} \end{array}$$

Figura 18: Producto entre una matriz y vector

En nuestro caso vamos a tratar con matrices dispersas. Las matrices dispersas son matrices de gran tamaño, en las que la mayoría de elementos son cero. Existen un gran número de algoritmos que relacionan las matrices dispersas, como por ejemplo: la suma, el producto, la transposición y factorizaciones como Cholesky, QR y LU entre otras. La gran ventaja de la utilización de estas matrices es que su uso está bastante extendido.

Estas matrices se representan de muchas formas, la forma más intuitiva es la que se denomina triplete. En ella tenemos tres vectores, uno para la fila en la que se sitúa el elemento no cero, otra para la columna y una tercera para el valor. De esta forma conseguimos ahorrar espacio, ya que almacenar todos los valores 0 es un gasto inútil de memoria, y a la larga también supone una pérdida de rendimiento.

Sin embargo, existen otro tipo de compresiones mucho más eficientes, aunque aumentan en su grado de complejidad. El formato que nosotros vamos a utilizar es el CSR (*Compressed Sparse Row*), o dicho de otra forma matriz dispersa comprimida por filas. En este caso también disponemos de tres vectores pero funcionan de distinto modo. A continuación vamos a mostrar una figura y su correspondiente compresión.

$\begin{pmatrix} 1 & 0 & 0 & 2 & 0 \\ 3 & 4 & 0 & 5 & 0 \\ 6 & 0 & 7 & 8 & 9 \\ 0 & 0 & 10 & 11 & 0 \\ 0 & 0 & 0 & 0 & 12 \end{pmatrix}$	<b>TRIPLETE</b>	
	VALOR	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
	FILA	(0, 0, 1, 1, 1, 2, 2, 2, 2, 3, 3, 4)
	COLUMNA	(0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4)
	<b>CSR</b>	
	VALOR	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
	COLUMNA	(0, 3, 0, 1, 3, 0, 2, 3, 4, 2, 3, 4)
	PUNTERO FILA	(0, 2, 5, 9, 11, 12)

Figura 19: Matriz dispersa, comprimida triplete y CSR

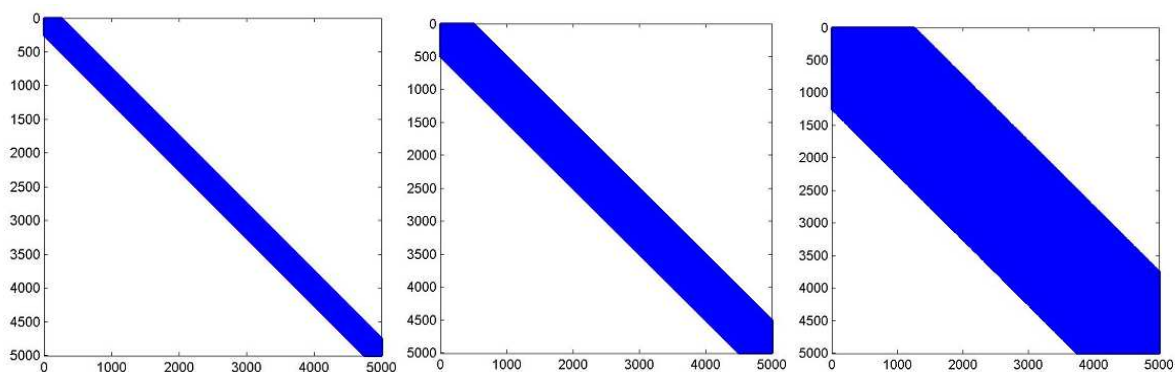
Ambos tipos difieren sólo en un vector. En el triplete como hemos dicho tenemos dos vectores para la coordenada y uno para el valor. En el caso de CSR tenemos uno para el valor, otro para la coordenada de la columna y el tercero contiene las filas en las que se encuentran los valores. Para calcularlo deberemos restar el valor n+1 menos el valor n y tendremos como resultado el número de elementos que se encuentran en la fila n. Por ejemplo, si restamos 2

(valor de  $n+1$ ), menos 0 (valor de  $n$ ), obtenemos que hay 2 elementos en la fila 0, y así sucesivamente. Esto es algo más complejo pero vamos a tener una gran ganancia en ahorro de espacio, ya que este vector reduce su tamaño, desde el número de elementos totales al número de filas que contiene la matriz. Puede parecer poco, pero cuando hablamos de matrices que tienen un millón de elementos en mil filas, estamos hablando de  $1.000.000 - 1.000 = 999.000$  elementos menos, que nos ahorramos. Cada elemento es un entero, luego en este caso concreto ahorramos aproximadamente 3,8 MBytes, que puede ser perfectamente el tamaño de una caché.

Hasta ahora sólo hemos hablado de matrices en forma de triplete y su compresión pero existen tres formatos principales de representación de estas matrices <sup>[38]</sup>: Harwell-Boeing <sup>[39]</sup>, MatrixMarket <sup>[40]</sup> (con este formato es con el que vamos a trabajar nosotros) y Matlab <sup>[41]</sup>. Cuando descarguemos alguna matriz dispersa siempre se encontrarán en alguno de estos tres formatos. En caso de requerir algún tipo de conversión recomendamos la utilización del conversor BeBOP Sparse Matrix Converter <sup>[42]</sup>, creado por Mark Hoemmen profesor de la Universidad de California, Berkeley. Con el cual contacté para plantearle cuestiones en el formato de estas matrices y al que quiero agradecer su ayuda para aclarar ideas sobre este asunto.

Para cargar las matrices y realizar el producto haremos uso de la librería del profesor de la Universidad de Florida, Timothy A. Davis <sup>[43]</sup>. El paquete viene asociado a un libro publicado también por él mismo, recomendable en caso de querer introducirse dentro del código <sup>[44]</sup>. También provee de una gran colección de matrices de distintos tipos y tamaños que puede ser utilizada gratuitamente y de la que haremos uso <sup>[45]</sup>. Dentro de este paquete podemos encontrar distintos tipos de matrices ordenadas por el tipo de valor que almacenan. Pueden ser reales, enteras o binarias, en el caso de reales, como su propio nombre indica la matriz almacena valores reales, igual sucede en el caso de las matrices enteras. Y en el caso de las binarias sólo tendremos dos vectores, que indican la posición donde se encuentra el elemento, asumiéndose que el valor de todos los elementos no cero, es siempre uno. Personalmente nosotros trataremos principalmente con matrices enteras y reales.

Además de este paquete de matrices bastante extenso, también contamos con un generador de matrices desarrollado en el marco de este proyecto. Este generador consigue crear matrices con distinto grado de dispersión, a partir del mismo número de elementos. Para ello le debemos pasar como parámetros de entradas, el número de filas, el número de elementos por fila y la dispersión de estos elementos dentro de la matriz en porcentaje, de esta forma obtendremos distintos conjuntos de matrices como los que vemos a continuación:





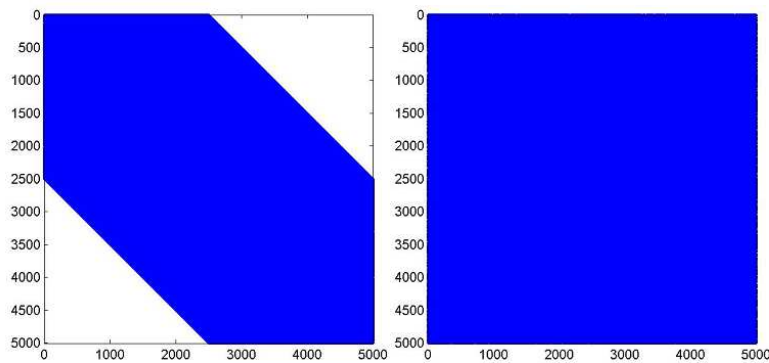


Figura 20: Matrices con grados de dispersión 1, 2, 5, 10 y 20 respectivamente, número de filas 5000, número de elementos por fila 500.

En la Figura 20 podemos ver cómo el generador de matrices crea distintos tipos de matrices con el mismo número de elementos. La gran dificultad de este generador es aumentar la dispersión de los elementos manteniendo el número de ellos constante.

Ambos tipos de matrices serán utilizados por la librería de Tim Davis y en concreto por la función `gaxpy`, que se encarga de multiplicar cualquiera de estas matrices por un vector. Para simplificar un poco la tarea el vector consiste en una sencilla progresión aritmética de aumento +1, que comienza en 1. Esto es 1, 2, 3, 4... y así hasta tanto elementos como número de columnas tenga la matriz elegida.

Al comienzo del desarrollo del benchmark surgió un pequeño problema en cuanto a los formatos de las matrices. El conjunto de matrices de Tim Davis, no son "0-based". Cuando decimos que una matriz es 0-based, decimos que la primera fila y la primera columna se identifican como 0. En este conjunto la primera fila y columna se identifican con 1, por ello tuvimos que realizar un pequeño conversor que se encargase de restar uno a todas las coordenadas para que los resultados fuesen coherentes. Además dentro de este conversor también existe una funcionalidad que se encarga de traducir las matrices generadas por el programa creado para este proyecto, a un formato entendible por la función de carga de Tim Davis.

Otro problema relacionado con la parte de la experimentación es que el código de Tim Davis está realizado basado en el formato de compresión CSC. Igual que CSR pero comprimido por columnas. Estos formatos están relacionados, puesto que si hacemos la traspuesta de la matriz y calculamos su CSC, es igual que si a la matriz original le aplicásemos el CSR. El caso es que CSC hace que exista un gran número de dependencias a la hora de realizar el producto. Veamos con más detalle estas dependencias.

En la Figura 21, que vemos a continuación, se observa claramente como en el reparto por columnas, hay una clara dependencia entre los distintos procesos. Todos ellos tienen en común el vector de escritura, por tanto, hay un alto grado de probabilidad de que el resultado de la operación sea distinto según qué procesos ejecuten primero. Debido a que OpenMP no proporciona semáforos para controlar este tipo de variables, esto introduce una complejidad debido a la necesidad de instrucciones de exclusión mutua para controlar el acceso a este vector común.

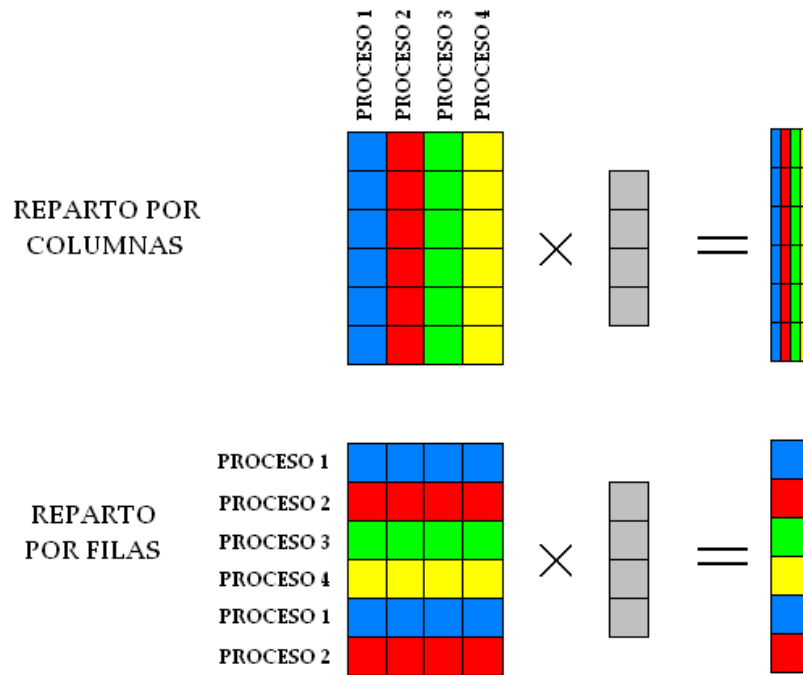


Figura 21: Reparto de la carga de producto de matriz-vector por filas y columnas

Para resolverlo decidimos pasar directamente a una multiplicación de matrices comprimidas por fila, y de esta forma las zonas de escritura no se solapan. Para ello fue necesario reescribir y modificar partes del código de Tim Davis y es por ello que para la ejecución de este benchmark es necesario tener la librería nuestra modificada. En caso de tomar la original si se utiliza la compresión por filas no funcionará. Esta parte queda detallada en el apéndice A: PAPI Guía de instalación y uso.

En esta técnica ambos procesadores deberían estar funcionando al 100%, pero no es así, dependiendo de la matriz con la que estemos tratando pueden llegar a tener una ocupación del 160%, es decir uno al 100% y otro al 60%. La única explicación que encontramos es que tengamos un cuello de botella en el bus, y éste no sea capaz de proveer de los datos suficientes al núcleo que está más ocioso. Para intentar aclarar esto realizamos una pequeña implementación, en la que se requerían más cálculos, con menos datos, y ambos procesadores alcanzaban el 100% de ocupación. En la Figura 22 vemos el comportamiento de los núcleos en el Core2Duo, para la función simple de producto matriz-dispersa vector y en ella veremos cómo los comportamientos son diferentes según el número de elementos no cero de la matriz:

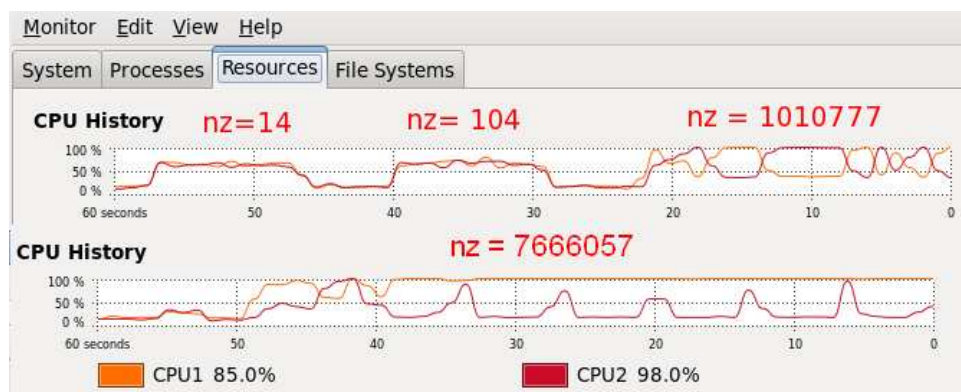


Figura 22: Ocupación de los núcleos en la prueba producto matriz-dispersa vector.

### 3.4 Técnica de optimización

Hasta ahora hemos hablado de diferentes pruebas que vamos a ejecutar en ambas plataformas, sin embargo, todavía no nos hemos explicado cómo vamos a conseguir que este benchmark sea adaptativo.

Las diferentes características de todos los sistemas hacen muy difícil que un benchmark sea capaz de adaptarse de forma eficiente a todas las plataformas existentes. Es por ello que hemos decidido realizar la búsqueda de esta adaptabilidad en el reparto de tareas a los distintos procesos. Realizaremos un estudio desde el punto de vista del tamaño de grano, para buscar el punto óptimo en el que una plataforma es capaz de ejecutar en un menor tiempo con un menor número de fallos las tareas que imponamos. En esta búsqueda del óptimo utilizaremos la técnica Simmulated Annealing y la Búsqueda N-aria.

#### 3.4.1 Simmulated Annealing

Simmulated Annealing es una técnica de búsqueda estocástica que se utiliza cuando el espacio de búsqueda no es bien conocido y no sigue una estructura sencilla, casos en los que el método de Newton no puede ser utilizado. En particular, esta técnica se utiliza frecuentemente para resolver problemas de optimización combinatoriales como el problema del viajante <sup>[46]</sup>.

Como sabemos en nuestro caso vamos a obtener gráficas de tiempos y de estadísticas de la CPU, que van a variar en función del tamaño de grano que vayamos a introducir. Nuestro objetivo será el de buscar un tamaño de grano concreto que sea capaz de minimizar los costes de ejecución.

El nombre de este algoritmo proviene del método de creación de algunos aceros o cristales. Mediante el cual los materiales se elevan a grandes temperaturas y, lentamente, se va enfriando hasta obtener las propiedades deseadas. El proceso de enfriado es clave para obtener los mejores resultados, ya que durante este proceso, las partículas se reordenan en su estado de mínima energía hasta que se obtiene un sólido con sus partículas acomodadas conforme a una estructura de cristal <sup>[47]</sup>.

La función tiene un estado de partida dado, a cada estado se le da una energía, esta energía es el valor de la función en ese punto. La probabilidad de dar un paso hacia otro estado viene dada por la distribución Boltzmann <sup>[48]</sup>.

Si nos adentramos un poco para calcular la probabilidad de que estemos en el estado “i” con energía “E<sub>i</sub>” a la temperatura “T” deberemos utilizar la siguiente fórmula:

$$P_T \{X = i\} = \frac{1}{Z(t)} \exp\left(\frac{-E_i}{k_B T}\right)$$

Figura 23: Fórmula de cálculo de probabilidad en Simulated Annealing

Donde “X” es la variable aleatoria que denota el estado actual y “Z” es una constante de normalización llamada función de partición. Se calcula como:

$$Z(T) = \sum_j \exp\left(\frac{-E_j}{k_B T}\right)$$

Figura 24: Fórmula de la constante Boltzmann

La parte que se encuentra dentro del sumatorio es la que se conoce como constante Boltzmann. El resultado final de esta ecuación probabilística es mayor o igual a cero. Y como no podía ser de otra forma la suma de todas las probabilidades es igual a uno.

Utilizaremos esta función porque de antemano sabemos que las gráficas generarán una forma de U en bastantes casos. Esto se debe a que cuando se realice un reparto con tamaño de grano fino, los procesos realizarán un gran número de cambios de contexto, ya que sus tareas finalizarán en breve; por otra parte, cuando el tamaño de grano sea muy grande el reparto estará desbalanceado y algunos procesos tendrán más carga que otros, con el consecuente aumento de fallos caché y aumento de tiempos.

En la Figura 25 vemos un ejemplo figurativo del funcionamiento de *Simulated Annealing* sobre una gráfica con forma de W.

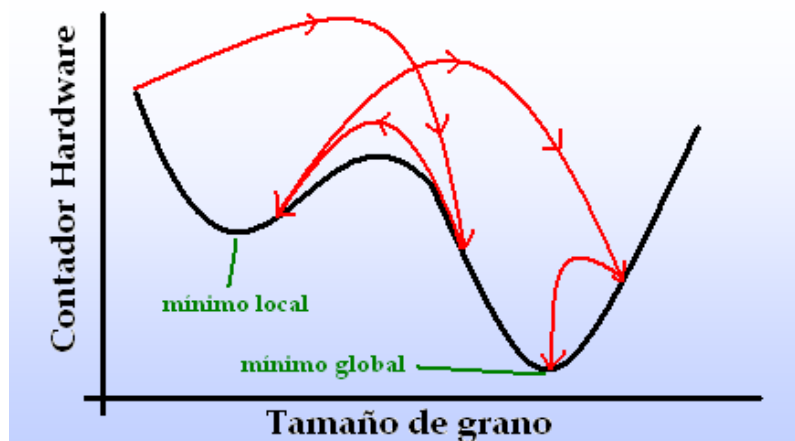


Figura 25: Explicación de *Simulated Annealing*

En este caso la función consta de un mínimo local y un mínimo global. Buscaremos que se produzca esta situación para poder testear la fiabilidad de este algoritmo. Al comienzo de la iteración en este caso se ha elegido como punto de partida el menor tamaño de grano que es 1. En las primeras iteraciones la temperatura inicial es muy alta y esto provoca que exista una mayor probabilidad de que se dé un paso. Además la idea de *Simulated Annealing* es que al comienzo de la ejecución los pasos dados sean más grandes para evitar caer en mínimos locales, y cuando se cree que se está en una zona de posible mínimo global disminuirá el tamaño del siguiente paso, para acercarse así de forma más suave al posible mínimo.

La implementación de esta técnica es muy sencilla, a pesar de la dificultad que parece tener en un principio. Usaremos una implementación ya existente y bastante extendida, nos referimos a la librería científica de GNU <sup>[49]</sup>, que contiene el código de *Simulated Annealing* para ser utilizado <sup>[50]</sup>.

En nuestro caso sólo deberemos añadir la implementación de tres funciones:

- Energía: es la función que devuelve la energía del estado correspondiente al que nos encontramos. En nuestro caso del benchmark la energía será el valor devuelto por uno de los contadores que seleccionamos, dada una entrada concreta (tamaño de grano). Esta función supondrá ejecutar por dentro el benchmark para obtener el resultado y recuperar la información para entregársela a *Simulated Annealing* y que siga iterando.
- Distancia: devolverá el valor absoluto de la diferencia de energías entre el nuevo estado y el anterior.
- Siguiente paso: función que calcula cuál será el siguiente paso a dar, si es que se decide dar internamente. Esta última decisión depende de la función probabilística que hemos visto.

Además de estas funciones también podremos modificar ciertos parámetros, que son definidos como constantes dentro del el código de *Simulated Annealing*. Estos son:

- Temperatura inicial: Es la temperatura con la que comienza a ejecutar, y ésta se utilizará para calcular la probabilidad de cambiar a otro estado en la primera iteración. En el resto de iteraciones se irá incrementando hasta alcanzar la temperatura final.
- Temperatura final: Es la temperatura límite, cuando se supera ésta la función probabilística determina que se ha finalizado y ninguna iteración más la precederá.
- Número de intentos: Además de realizar un límite por temperatura también se puede limitar por número de intentos. Dentro de cada intento también podemos hablar de distinto número de iteraciones. De tal forma que las iteraciones totales realizadas serán el número de intentos multiplicado por el número de iteraciones.
- Paso o salto:
  - Tamaño de paso: indicará el tamaño del salto medio en una iteración.
  - Paso máximo: salto máximo que puede darse. Será igual a la diferencia entre el máximo tamaño de grano que elijamos para limitar la función y el mínimo tamaño de grano posible (uno).
  - Paso mínimo: mínimo salto posible (uno).

### 3.4.2 Búsqueda N-aria

Otra de las técnicas de optimización que vamos a utilizar va a ser la que hemos autodenominado la Búsqueda N-aria. El algoritmo fue creado por nosotros para este proyecto, con la finalidad de compararlo frente a *Simulated Annealing*. Esta técnica tratará de buscar el mejor tamaño de grano para cualquiera de las plataformas con las que vayamos a trabajar. Se podría definir como una mezcla entre la búsqueda binaria y la búsqueda en haz.

El algoritmo de búsqueda binaria consiste en dividir el espacio de búsqueda (ordenado) en dos partes, por su elemento central y, a partir de ahí, busca si el elemento se encuentra en la parte primera o segunda del conjunto. En nuestro caso vamos a tener un espacio de búsqueda conformado por los tamaños de grano de 1 a 1024. Con la Búsqueda N-aria vamos a dividir este espacio de búsqueda en “N” partes y vamos a realizar observaciones de estas divisiones o puntos limítrofes. Cuando tengamos los valores calcularemos el punto medio de las zonas que hemos delimitado.

El siguiente paso será seleccionar las “K” mejores zonas, para realizar un estudio sobre ellas. Para seleccionar estas zonas dividiremos entre dos la suma de los valores que calculamos delimitando dichas zonas y éste valor será el que nos indicará cuales son las mejores zonas. Volviéndolas a dividir en “N” partes. Si recordamos el algoritmo en haz, cuando tenemos un árbol de búsqueda nos encargamos de seleccionar los “K” mejores descendientes para continuar el estudio. De esta forma recorreremos el espacio de búsqueda en anchura hasta llegar a un nivel en el que las secciones elegidas no puedan ser divididas en más partes.

Por tanto, para la ejecución de la búsqueda binaria necesitaremos definir las siguientes constantes:

- Secciones o N: número de partes en las que dividiremos el espacio de búsqueda para realizar un estudio de él.
- Número máximo de ejecuciones: limitará el número de llamadas que se realizarán al benchmark.
- Exploración o K: número de secciones que desearemos explorar.

Vamos a ver un ejemplo del funcionamiento, tomando un N=10 y un K=3:

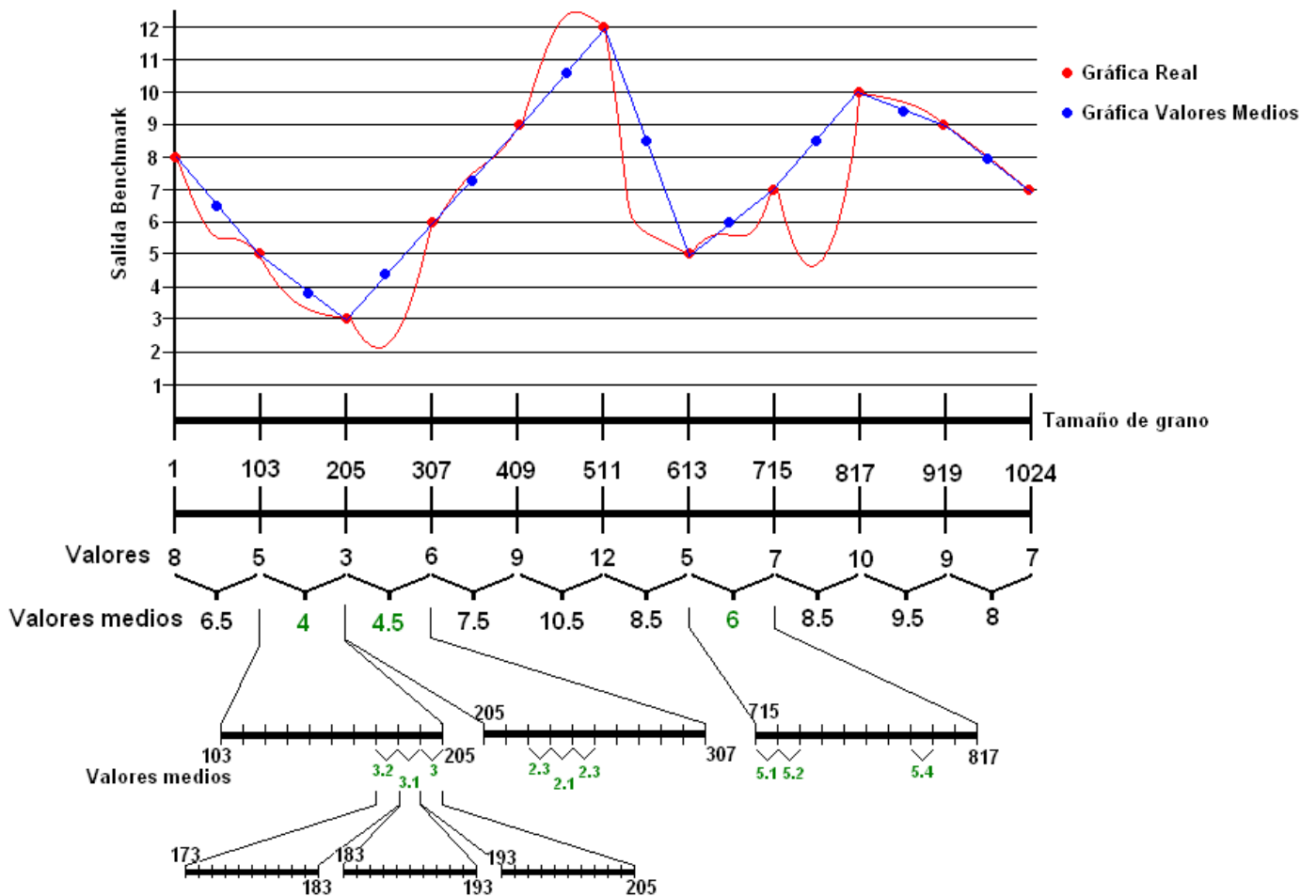


Figura 26: Ejemplo de Búsqueda N-aria

En este ejemplo dividimos la gráfica en diez secciones, y en cada parte limítrofe tomamos los valores de la función (en la gráfica se representan mediante los puntos azules). El siguiente paso es calcular los valores medios de las secciones. Las mejores secciones son la segunda (103-205), la tercera (205-307) y la séptima (613-715). Estas serán las secciones elegidas para realizar el estudio de siguiente nivel.

Como podemos comprobar esta búsqueda tiene un problema, y es que cuantas más oscilaciones tenga, más difícil será encontrar el mínimo global. Este es el caso de la sección octava (715-817), que a pesar de tener un mínimo local, pasa a ser totalmente descartada. Para solucionar este tipo de problemas lo único que podríamos hacer es aumentar el número de secciones (N) y un aumento de zonas a explorar (K), de esta forma, tendríamos un mayor conocimiento de la función. Aunque esto supondría un mayor coste de computación, puesto que estamos aumentando el número de iteraciones por nivel.

El siguiente paso es analizar las tres secciones elegidas. El procedimiento será el mismo, aunque cabe destacar que no se bajará al siguiente nivel hasta que todas las nuevas secciones a estudiar hayan sido completadas. Finalmente el buscador parará cuando alcance el límite de iteraciones indicado, o cuando las secciones que vayamos a analizar tengan menos elementos que el número de secciones en las que estamos dividiendo estos grupos (N). En el caso concreto de la gráfica el mejor mínimo se encuentra entre los límites 205 y 307 y además vemos cómo ha seleccionado las secciones más interesantes para seguir explorando. Aunque

en la gráfica no hemos desarrollado por completo todas las secciones, en las tres siguientes encontrará el mínimo global (2,1 aproximadamente).

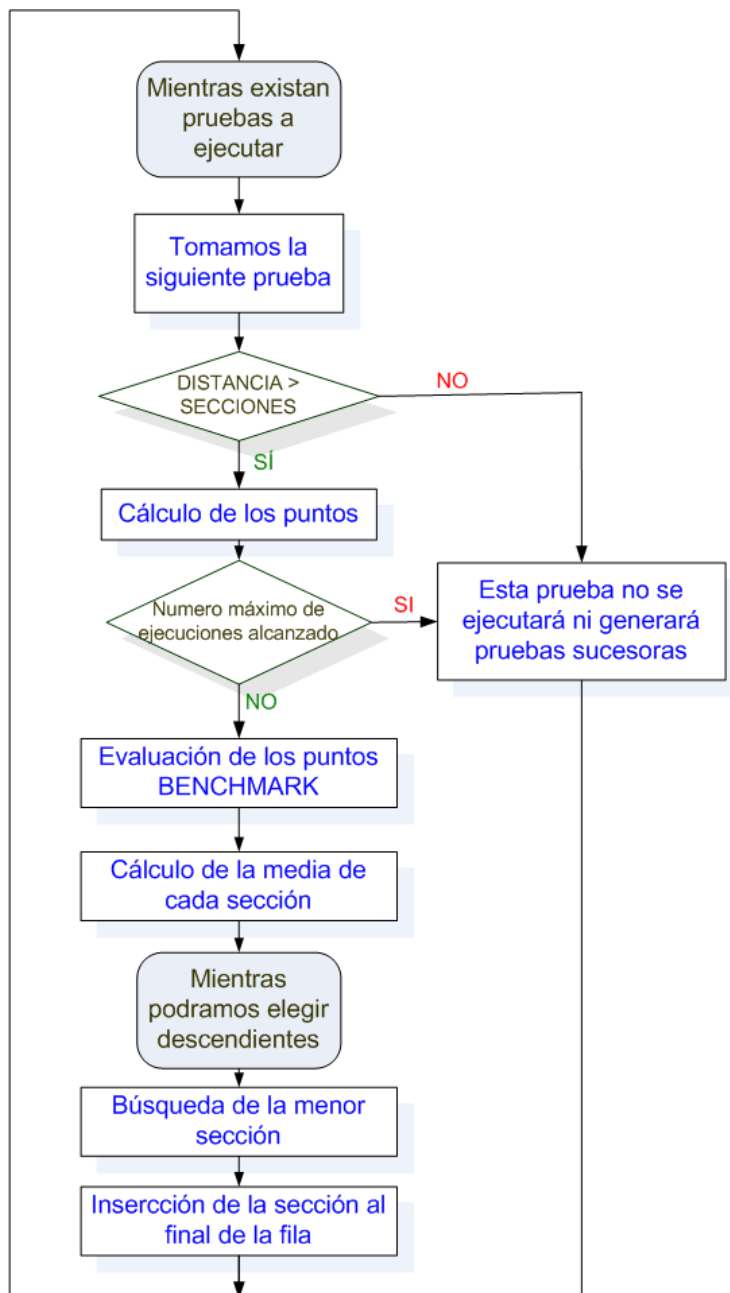


Figura 27: Diagrama de Búsqueda N-aria

Vamos a ver un pequeño diagrama de código para que se entiendan mejor los pasos a seguir de cara a futuros programadores que deseen añadir alguna mejora a esta búsqueda. Básicamente consiste en un bucle en el que internamente iremos añadiendo pruebas, comenzaremos con la prueba original y continuaremos hasta que:

- Distancia > Secciones: esto quiere decir que pararemos cuando el límite superior de la sección menos el límite inferior de la sección de cómo resultado un número menor al número de secciones.
- Máximo de ejecuciones alcanzado: no podremos ejecutar el benchmark más veces de las que indiquemos con la constante definida.

Este tipo de búsqueda es mucho más sencilla que *Simulated Annealing* y vamos a poder comparar de forma directa cuál de los dos métodos funciona mejor. Aunque sí que es cierto que los métodos ofrecerán mejores resultados para ciertos tipos de gráficas. Por ejemplo, en el caso de la Búsqueda N-aria el mejor caso se producirá cuando el mínimo global se encuentre en una de las divisiones que realicemos en la primera iteración. El peor caso será cuando la sección en la que se encuentra el mínimo global quede descartada y no se siga explorando.



## 4.- Experimentos

---

En este cuarto apartado nos centraremos en experimentar con el código creado. Hablaremos también sobre qué parámetros se van a evaluar y cómo van a ser tratados los resultados arrojados por el benchmark en las distintas plataformas.

### 4.1 Definición de parámetros de la arquitectura a evaluar

Para la selección de los parámetros vamos a utilizar contadores que se encuentren en ambas plataformas, de esta forma podremos realizar una comparación directa. Sin embargo, no abordaremos más de cinco contadores, puesto que a lo largo del estudio realizaremos bastantes pruebas y cada contador que añadamos, aparte de poder requerir alguna ejecución de más, también requiere un análisis de lo que está sucediendo. Es por ello que será muy importante la selección de estos contadores, ya que serán claves en el análisis del benchmark y de los resultados a analizar.

Si bien es cierto que cuando presentamos los contadores hablamos de tres grandes grupos: contadores de caché, contadores de instrucciones y contadores de ciclos. En el caso de nuestro benchmark nos centraremos principalmente en los contadores caché haciendo incidencia en la parte de datos, obviaremos los contadores de instrucciones y tomaremos algún contador de ciclos. Nuestro programa va a tener una fuerte dependencia de datos, y las instrucciones se repetirán a lo largo de las iteraciones, por lo que la parte clave, o la que más problemas dará a la CPU, será la de carga de datos (y no la de carga de instrucciones).

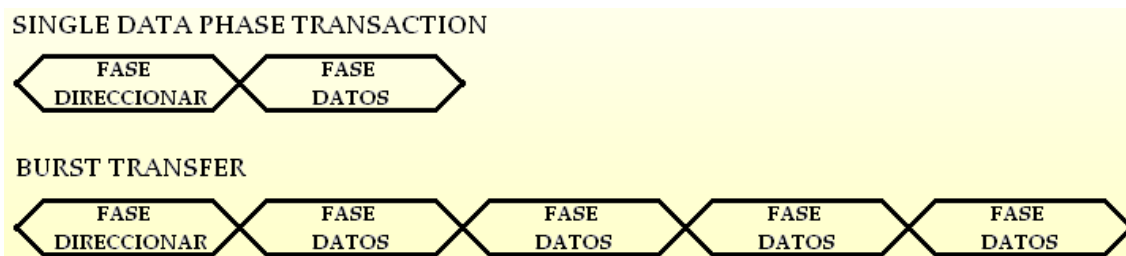
Los contadores seleccionados, comunes a ambas plataformas, son:

- PAPI\_L1\_DCM: Indica el número de fallos de datos que se producen en la caché de nivel 1. Disponible en ambas arquitecturas.
- PAPI\_L2\_DCM y L2\_LINES\_IN:BOTH\_CORES: El primer contador lo utilizaremos en el Opteron y el segundo en el Core2Duo. Esto se debe a que el primer contador en el Opteron es derivado y sabemos que este tipo de contadores suele dar problemas. El segundo contador a pesar de ser nativo, nos acabará mostrando la misma información que PAPI\_L2\_DCM.
- PAPI\_TLB\_DM: Refleja el número de fallos de datos que se producen en la TLB.
- PAPI\_TOT\_CYC: Total de ciclos de ejecución necesarios para ejecutar la prueba.

También haremos uso de un contador más que nos dé visión sobre lo que está ocurriendo en el bus de comunicaciones entre la memoria principal y la memoria caché. En este caso no existe un contador único común a ambas plataformas por lo que haremos uso de:

- BUS\_TRANS\_BRD:BOTH\_CORES: Contador que utilizaremos para leer estadísticas del bus en el Core2Duo. Este contador devuelve las transacciones de bus de lectura en ráfaga (burst read bus transactions). Dentro de los tipos de transacción de datos en el bus diferenciamos dos. El single data phase transaction en el que la cantidad de datos a mandar es muy pequeña y simplemente requerimos de una fase de

direccionamiento y una fase de transferencia de datos. En el “burst read transactions” la cantidad de datos a mandar es mucho mayor y, por tanto, la fase de direccionamiento va seguida de varias fases de datos en las que se manda la información. En nuestro caso la mayoría de las transacciones serán del segundo tipo, puesto que queremos saturar las memorias cachés para intentar evitar la reutilización de memoria. Veamos un ejemplo de ambos tipos de transacciones:



*Figura 28: Single data phase transaction vs burst transfer*

- CPU\_IO\_REQUESTS\_TO\_MEMORY\_IO:CPU\_TO\_MEM: Este contador nativo reflejará las peticiones que haga la CPU a la memoria para solicitar información. Utilizaremos este contador para ver el estado del bus de memoria. A mayor número de peticiones mayor saturación de éste.

#### 4.1.1 Selección dinámica de contadores

Durante el desarrollo de la herramienta fuimos topando con distintos errores a la hora de utilizar ciertos contadores no soportados por la plataforma, o con problemas de incoherencia de resultado. Éste último caso se produce cuando se utilizan más contadores de los soportados por la plataforma. En ocasiones cuando añadimos más contadores de los soportables, PAPI no refleja ningún tipo de error y, cuando recogemos los resultados tenemos valores del orden de  $10^{18}$ .

PAPI ya avisa de esto en su manual <sup>[51]</sup>, y propone como solución la utilización de la multiplexación. Sin embargo, en la plataforma en la que estamos implementando el benchmark no soporta la multiplexación, puesto que arroja errores al añadir eventos. Por tanto, la única salida que nos quedaba es probar directamente los contadores, ver qué resultados ofrecían y de esta manera, crear distintas rondas de pruebas clasificando los contadores en pequeños grupos de dos o tres, que pudiesen ejecutarse juntos y cuyos resultados fuesen coherentes.

Como explicaremos más detalladamente en el manual del usuario-programador, los contadores que debe tomar el benchmark vienen dados en un fichero de texto. Al ser una tarea tediosa probar los contadores y ver qué resultados ofrecen para luego ver si se pueden usar o no, hemos creado un programa capaz de analizar los contadores de tal forma que la salida que nos va a ofrecer es una lista de contadores y su compatibilidad a la hora de ser ejecutados. Esto es simplemente los conjuntos de pruebas que podrán ser ejecutadas, de forma secuencial, hasta tener todas las estadísticas requeridas de una prueba.

Mostramos en el siguiente cuadro de texto la salida de este testeador de eventos:

```

name=PAPI_L1_DCM, counter[0]=1
name=L2_LINES_IN:BOTH_CORES, counter[1]=1
name=PAPI_L3_DCM, counter[2]=-7
name=BUS_TRANS_BRD:BOTH_CORES, counter[3]=2
name=PAPI_TLB_DM, counter[4]=2
name=PAPI_TOT_CYC, counter[5]=2
num_tests=2

```

Código 3: Salida del testeador de contadores

En la salida se muestran los distintos contadores que hemos introducido en el fichero que se pasa como parámetro. Como podemos comprobar tenemos un total de dos ejecuciones, una primera en la que irían los contadores PAPI\_L1\_DCM y L2\_LINES\_IN:BOTH\_CORES, luego tenemos un evento que no podemos usar (PAPI\_L3\_DCM), en el que PAPI nos devuelve un error -7 (PAPI\_ENOEVT), el evento hardware no existe para esta plataforma. Y tenemos una última prueba en la que podemos englobar tres contadores: BUS\_TRANS\_BRD:BOTH\_CORES, PAPI\_TLB\_DM y PAPI\_TOT\_CYC.

Para poder testear estos contadores se realiza un pequeño bucle que suponga un mínimo de computación, para que al leer los contadores de nuevo no sean 0. Esta prueba se ejecuta al principio de todas las ejecuciones del benchmark, para comprobar que los contadores que introduzcamos en el fichero de texto son correctos para dicha plataforma y para minimizar el número de pruebas globales a realizar.

Es evidente que el orden en el obtengamos las salidas de los contadores va a ser totalmente dependiente del orden en el que incluyamos los contadores en el fichero, debido a que el programa tomará los distintos contadores siguiendo el orden de este fichero. Pero, para entender mejor el funcionamiento del sistema, vamos a ver un diagrama de su funcionamiento:

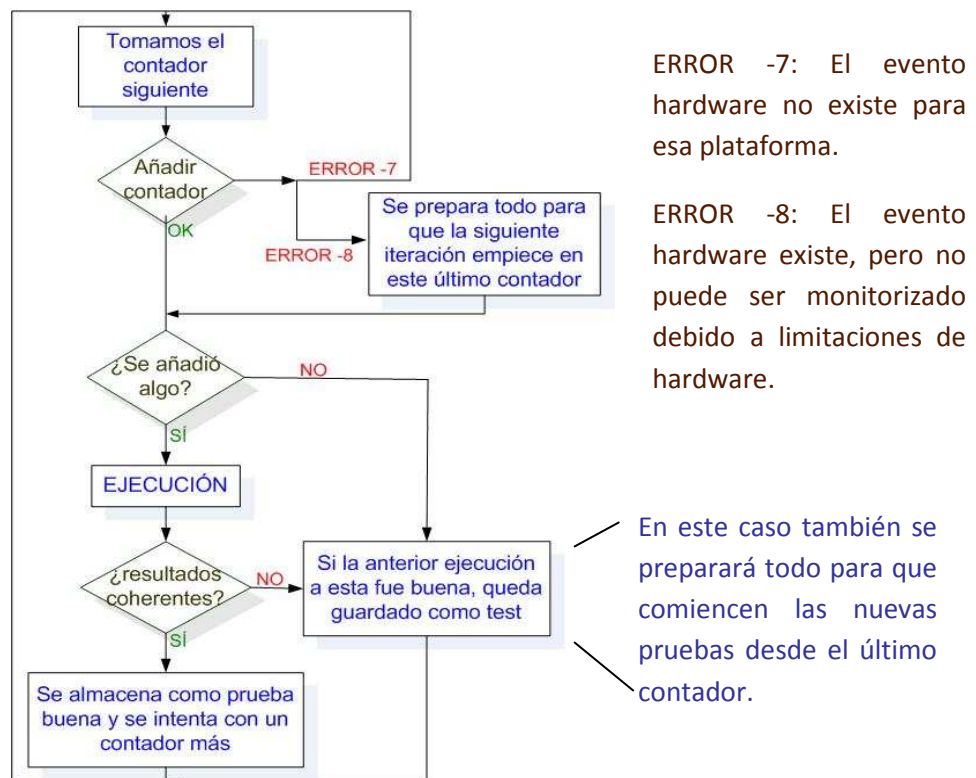


Figura 29: Diagrama de funcionamiento del testeador de evento

Prevía a la entrada en el diagrama se realiza una comprobación general de los contadores disponibles en la plataforma, si dan fallo se almacena el valor -7 dentro de un array de información.

El diagrama comienza básicamente tomando un contador, realiza el test y si funciona intenta meter otro contador, así hasta que no se puedan añadir más contadores. En caso de que no se pueda añadir el contador porque no exista para esa plataforma (-7), se ignora y se pasa al siguiente. En caso de no se puedan añadir más debido a limitación de hardware (-8), se deja todo preparado para que en la siguiente prueba se comience desde cero, tomando este contador, que falló en la última prueba, como primero de la nueva ronda. Después de realizar los test comprobamos que los valores de salida no sean mayores que  $10^9$ , en caso de ser mayores, se considera resultados incoherentes y se dejará todo preparado para que se comience a testear desde este último contador, que no falló a la hora de ser añadido pero provocó que los contadores diesen resultados incoherentes.

Es importante saber que, además de estar incorporado dentro del benchmark, también existe una versión ejecutable fuera de ella para poder comprobar de antemano qué contadores son compatibles para esa plataforma, y cuáles de ellos pueden ser agrupados en una misma prueba. Recomendamos acceder al Apéndice C para consultar cualquier duda.

## 4.2 Metodología para evaluar los parámetros de la arquitectura

Para evaluar los parámetros de la arquitectura primero vamos a tratar la estructura de salida del benchmark.

En la salida de una prueba simple tendremos:

```
PAPI_L1_DCM 49298
L2_LINES_IN:BOTH_CORES 1097
Output = 392822784000.000000,   Execution Time = 3.406823, cache_output =-4295.090000
BUS_TRANS_BRD:BOTH_CORES 42645
PAPI_TLB_DM 855
PAPI_TOT_CYC 6444569
Output = 392822784000.000000,   Execution Time = 3.402157, cache_output =-4029.070000
TIME 3.402157
```

*Código 4: Salida de prueba simple*

En ella vemos los distintos contadores que hemos utilizado y su correspondiente resultado. Output en este caso muestra la salida de la prueba, en este caso la suma de valores de una prueba de escritura con stride. Esta salida es muy importante ya que si no se incluyese el compilador podría llegar a ignorar todo el código, puesto que al no mostrar ninguna salida deduciría que la ejecución del código sería innecesaria. Es por ello que en todas las pruebas es obligatorio sacar la salida por pantalla.

Después del output vemos el tiempo de ejecución devuelto por PAPI y la salida del resultado de limpiar la caché.

Limpiar la caché es algo necesario en todas las pruebas, de esta forma nos aseguramos de que los datos no se encuentren en la caché a la hora de realizar una siguiente prueba. Para limpiar la caché rellenamos un vector de tamaño superior a la caché con valores

pseudoaleatorios, que dependen de la hora actual, para calcularlos utilizamos las funciones `gettimeofday()` y `srand()`. La pretensión era que `cache_output` devolviese valores cercanos al cero, trabajando con valores entre -1 y 1, sin embargo al depender de la fecha actual los valores tienen cierta similitud, aunque esto es suficiente para borrar la caché.

Tendremos una salida de `Output|Time|CacheOutput` por cada prueba que se realice, como dijimos la multiplexación se implementa manualmente, y cada salida de este tipo representa una ejecución con el conjunto de contadores que aparecen encima de dicha salida.

En las salidas de los scripts veremos en la primera fila una pequeña cabecera de resumen mostrando el tiempo y los contadores que fueron utilizados finalmente en el benchmark. Esta fila es interesante, puesto que no todos los contadores que hayamos introducido en el fichero de carga van a poder ser utilizados. En ella sólo aparecerán los contadores que están disponibles en la plataforma, y aparecerán en el orden indicado por el testeador de contadores.

Después de esta primera fila pasaremos a tener toda la información. Ésta se separará en espacios para poder ser tratada por una hoja de cálculo y así podrá ser analizada de una forma más cómoda.

Dentro de las distintas pruebas existentes hemos generado dos scripts para poder ejecutar las pruebas variando ciertos parámetros:

- Script de filas (*rows*): Este script servirá para ejecutar la prueba de escritura con stride con distintos tamaños de filas. Por tanto, después de la fila de cabecera aparecerán tantas filas, como pruebas queramos realizar con distintos tamaños de filas.
- Script de tamaño de grano (*chunk*): La función de este script es similar, puesto que es capaz de ejecutar distintas pruebas probando diferentes tamaños de grano. En este caso el script está dirigido a las pruebas de *false sharing* y a la de producto matriz-dispersa vector.

En el siguiente cuadro de texto mostramos un ejemplo de salida del benchmark para una pequeña prueba en la que se escribe en la matriz por filas (sin stride):

Rows	Time	PAPI_L1_DCM	L2_LINES_IN:BOTH_CORES	BUS_TRANS_BRD:BOTH_CORES	PAPI_TLB_DM
1	1.654751	49215	1647	50133	775
2	1.632102	49217	1731	50158	776
4	1.661810	49217	2118	51469	826
8	1.658812	49221	1903	51433	826
16	1.630318	49228	1557	50210	774
32	1.743056	49246	1496	50126	827

*Código 5: Salida benchmark modo rows, prueba matriz normal*

Según la salida que observamos vemos que se ejecutaron 6 pruebas para 6 matrices con distinto número de filas (1, 2, 4, 8, 16, 32), utilizando cuatro contadores hardware (PAPI\_L1\_DCM, L2\_LINES\_IN:BOTH\_CORES, BUS\_TRANS\_BRD:BOTH\_CORES, PAPI\_TLB\_DM). La segunda columna muestra el tiempo de ejecución.

### 4.3 Metodología de análisis de los resultados

Como hemos dicho en el apartado anterior, los resultados se ofrecerán separados por espacios, así al copiarlos a una hoja de cálculo podremos dividirlos en diferentes columnas, utilizando estos espacios como caracteres separadores.

Pero antes de realizar este tratamiento los datos pasarán por un script más. Debido al gran número de datos que vamos a obtener, y a que a pesar de que los resultados que obtenemos de una misma prueba son los mismos, los resultados que ofrecen los contadores van a ser diferentes. Hemos decidido dar la posibilidad al usuario de ejecutar una misma prueba “n” veces. De esta forma vamos a calcular la media y la desviación típica de estos valores, teniendo así una medida mucho más fiable. Este script del que estamos hablando se encargará de añadir a un fichero los valores de las distintas pruebas realizadas y finalmente llamará a un programa de estadísticas. Este programa no sólo se encarga de realizar la media y la desviación típica de los valores tomados, sino que además ignora alguno de los valores que se alejan más de la media, para evitar así que las medidas resultantes sean poco precisas. Por lo tanto, se aconseja en la utilización del benchmark que el número de repeticiones sea mayor o igual a 5. En la siguiente salida podemos observar los distintos valores para nueve pruebas con distintos tamaños de grano, los resultados corresponden al tiempo y a dos contadores (PAPI\_L1\_LDM L2\_LINES\_IN:BOTH\_CORES). Entre paréntesis se observa la desviación típica de los valores.

Chunk	TIME	PAPI_L1_LDM	L2_LINES_IN:BOTH_CORES
1	0.000405	(0.000002)	47981.600000 (187.353783) 1086.900000 (24.704048)
2	0.000407	(0.000002)	44687.200000 (562.924826) 1280.300000 (32.400772)
4	0.000401	(0.000001)	38756.500000 (251.650253) 1793.300000 (48.108315)
8	0.000393	(0.000003)	35407.200000 (173.914807) 2682.700000 (93.875503)
16	0.000364	(0.000002)	32277.900000 (101.649840) 3769.400000 (56.359915)
32	0.000337	(0.000002)	29052.800000 (95.657514) 2906.800000 (108.209796)
64	0.000327	(0.000004)	26617.700000 (48.083365) 2372.100000 (75.376986)
128	0.000323	(0.000002)	25312.100000 (123.495304) 2030.700000 (97.344800)
256	0.000328	(0.000004)	24465.100000 (36.352304) 1598.500000 (36.373754)

*Código 6: Salida final benchmark, prueba producto matriz-dispersa vector*

En cuanto al producto matriz-dispersa vector, en ocasiones realizaremos una normalización de los datos. Este se debe a que compararemos matrices con distintos tamaños y no es correcto que comparemos los contadores de matrices muy grandes con matrices pequeñas, sin previa normalización.

Para facilitar la tarea de comprensión de los resultados, éstos se compararán mediante gráficas, en las que daremos nombre a ambos ejes de coordenadas y mostraremos los resultados utilizando la notación científica. Intentaremos mostrar una visión de lo que puede haber ocurrido en el interior de la CPU. Decimos intentaremos porque es una tarea difícil de determinar con precisión. Dentro de un microprocesador son muchas las variables que pueden afectar y tener un conocimiento general del funcionamiento de éste es clave para hacer una buena aproximación de los resultados. Es importante saber que se le da la opción al usuario de modificar los dividiéndolos entre un orden de magnitud en caso de que se deseen reducir las estadísticas a un orden más pequeño.

Cada prueba que se realiza será identificada con un número y todas las salidas quedarán almacenadas y podrán ser consultadas en la entrega digital de este proyecto. En la siguiente tabla mostramos un resumen de los experimentos realizados y los respectivos apartados en los que los podremos encontrar:

Nº Experimento	Apartados	C2D	OPT	ES	FS	PMV	NH	HP	PF	SA	BN
1	4.4.1	✓		✓							
2	4.4.1, 4.1.1.1, 4.4.3	✓		✓					✓		
3	4.4.1.1	✓		✓					✓		
4	4.1.1.1	✓		✓					✓		
5	4.1.1.1	✓	✓	✓					✓		
6	4.4.2, 4.4.3		✓	✓							
7	4.4.2		✓	✓							
8	4.5.1	✓			✓						
9	4.5.1, 4.5.1.1, 4.5.3, 4.7	✓			✓				✓		
10	4.5.1.1	✓			✓				✓		
11	4.5.1.1	✓			✓				✓		
12	4.5.1.1	✓			✓				✓		
13	4.5.2, 4.5.3		✓		✓						
14	4.5.2		✓		✓						
15	4.6.1.1, 4.6.1.3		✓			✓					
16	4.6.1.2, 4.6.1.3		✓			✓					
17	4.6.2.1, 4.6.2.3	✓				✓	✓				
18	4.6.2.2, 4.6.2.3		✓			✓	✓				
19	4.6.3		✓			✓					
20	4.6.4		✓								
21	4.7		✓					✓			
22	4.8	✓	✓			✓				✓	✓

Tabla 4: Resumen de experimentos.

Leyenda:

C2D: Intel Core2Duo

OPT: AMD Opteron

ES: Escritura con stride → Escritura en una matriz por columnas, realiza saltos de memoria.

FS: False sharing → Dos o más procesos escriben en zonas contiguas de memoria.

PMV: Producto matriz-dispersa vector → Producto entre una matriz dispersa y un vector

NH: Número diferente de hilos

PF: Prefetching Desactivado

SA: *Simulated Annealing*

BN: Búsqueda N-aria

Como indica la leyenda las pruebas se muestran separadas en las diferentes arquitecturas y los distintos métodos empleados. Todos estos datos se adjuntan junto con la documentación en la carpeta resultados. En la siguiente tabla veremos la misma información pero desde el punto de vista de los apartados.

Apartados	Nº Experimento	C2D	OPT	ES	FS	PMV	NH	HP	PF	SA	BN
4.4.1	1, 2	✓		✓							
4.1.1.1	2, 3, 4, 5	✓		✓					✓		
4.4.2	6, 7		✓	✓							
4.4.3	2, 6	✓	✓	✓							
4.5.1	8, 9	✓			✓						
4.5.1.1	9, 10, 11, 12	✓			✓				✓		
4.5.2	13, 14		✓		✓						
4.5.3	9, 13	✓	✓		✓						
4.6.1.1	15	✓				✓					
4.6.1.2	16		✓			✓					
4.6.1.3	15, 16	✓	✓			✓					
4.6.2.1	17	✓				✓	✓				
4.6.2.2	18		✓			✓	✓				
4.6.2.3	17, 18	✓	✓			✓	✓				
4.6.3	19		✓			✓					
4.6.4	20		✓			✓					
4.7	9, 21	✓			✓			✓			
4.8	22	✓	✓			✓				✓	✓

Tabla 5: Resumen de apartados.

Leyenda:

C2D: Intel Core2Duo

OPT: AMD Opteron

ES: Escritura con stride → Escritura en una matriz por columnas, realiza saltos de memoria.

FS: False sharing → Dos o más procesos escriben en zonas contiguas de memoria.

PMV: Producto matriz-dispersa vector → Producto entre una matriz dispersa y un vector

NH: Número diferente de hilos

PF: Prefetching Desactivado

SA: *Simulated Annealing*

BN: Búsqueda N-aria



### 4.4 Escritura con stride

Llegados a este punto ya sabemos cómo funciona el algoritmo de escritura con stride, los siguientes pasos serán probar este test en ambas plataformas, y además realizaremos una comparativa de las mismas.

#### 4.4.1 Intel Core2Duo

Para realizar las primeras pruebas sobre esta arquitectura vamos a utilizar dos tamaños distintos de matrices, uno en el que todavía quepa en la memoria caché y otro en la que rebase este tamaño. La memoria caché de nivel 2 de esta arquitectura es de 2MB, y cada elemento de la matriz es un entero, ocupa 4 Bytes. Para calcular el número de elementos simplemente dividiremos el tamaño que esperamos que ocupe la matriz entre 4. Luego tendremos los siguientes casos:

- Matriz de 1.5MB  $\rightarrow$  luego número de elementos es igual a  $(1.5 \cdot 1024 \cdot 1024) / 4 = 393216$  elementos. Experimento 1.
- Matriz de 3MB  $\rightarrow (3 \cdot 1024 \cdot 1024) / 4 = 786432$  elementos. Experimento 2.

Se espera que los resultados obtenidos en la matriz de 1.5 sean mucho mejores que en el caso de la de 3MB, puesto que al entrar completamente en caché no deberán producirse tantos fallos a la hora de cargar los datos.

En las siguientes gráficas mostraremos la salida del benchmark. Se ejecutarán las mismas pruebas 15 veces y tomaremos la media de cada uno de los contadores y además del tiempo.

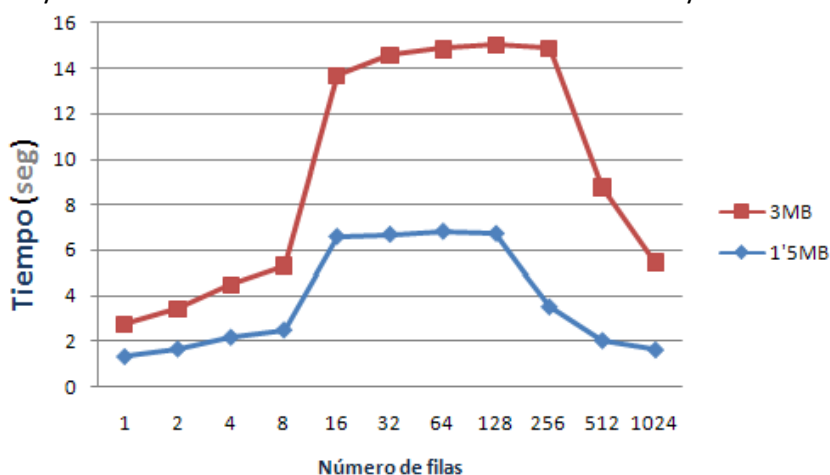


Figura 30: Tiempos para experimentos 1 y 2 [ES][Core2Duo]

En las primeras ejecuciones vemos cómo lentamente aumentan los tiempos, siendo el de la matriz grande aproximadamente el doble que en el caso de la pequeña. Además se produce un gran salto al pasar de 8 a 16 filas. En la previsión realizada teóricamente, cuando desarrollábamos el software no pensábamos que volviese a bajar, ya que no tiene sentido que al aumentar el salto suceda esto. Podríamos pensar que si dividimos en muchas filas la matriz podría llegar un momento en el que el tamaño de la fila fuese tan pequeño que al cargar los datos de una fila estuviésemos cargando los de la siguiente. A continuación un ejemplo gráfico:

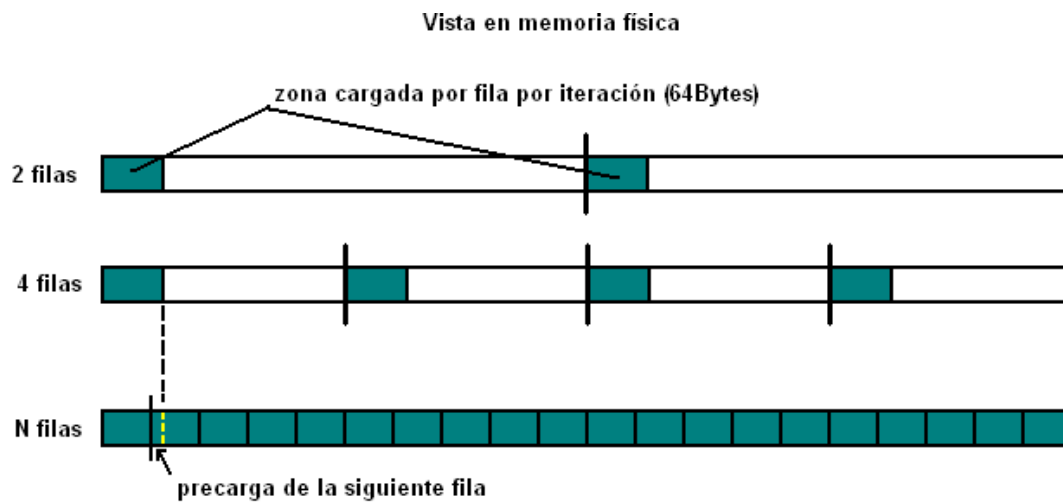


Figura 31: Caso en el que dos filas puedan solapar en un mismo bloque

Pero para que esto llegue a suceder, en el caso de la matriz de 1.5MB tendríamos que hablar de un número de filas superior a  $1.5\text{MB}/64\text{B} = 24576$  filas, algo que no se alcanza.

La única opción que parece dar explicación a esta bajada podría ser el *prefetching* que comienza a funcionar mejor para los casos en los que el número de filas es 512 o 1024, que para los casos de 16, 32 y 64 filas. Pero como veremos en el siguiente apartado, 4.4.1.1, esto no es así.

Para intentar explicar lo que está sucediendo se creó un simulador mediante Matlab, en el cuál mediante código se representa la memoria caché y se trata de dar explicación a la bajada de tiempos. En este simulador intentaremos simular el efecto de llenado de una caché, y veremos que mediante la política de reemplazo LRU (se reemplaza el bloque que fue utilizado hace más tiempo), aumentará el número de fallos. En la figura que vemos a continuación tenemos la salida del simulador, en el eje Y tenemos el número de fallos en L1, y en el eje X el logaritmo en base 2 del número de filas.

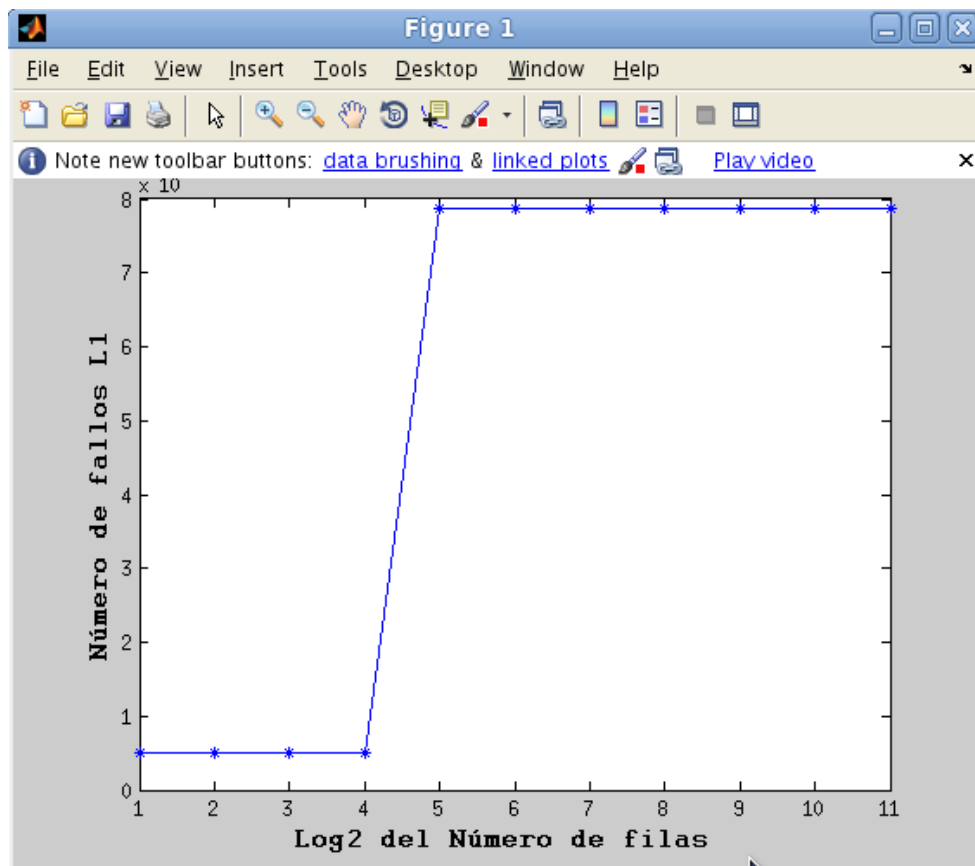
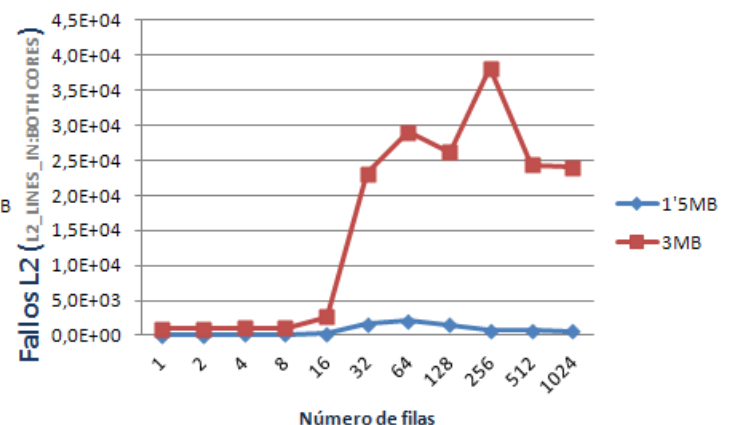
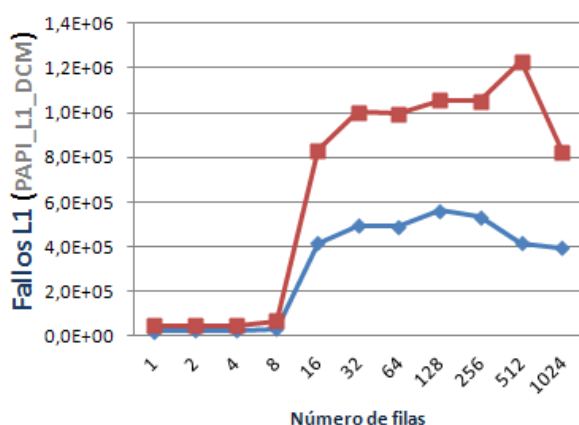


Figura 32: Salida del simulador de escritura con stride

En la salida (Figura 32) queda reflejada la subida de fallos pero no es capaz de explicar porqué se vuelve a producir la bajada. Es un caso extraño ya que anticipamos que este efecto no se producirá en el Opteron. Una de las posibles ideas es que algún proceso esté interfiriendo en la memoria caché durante la ejecución de la prueba. Pero después dejar en el Core2Duo ejecutando sólo los procesos necesarios, el resultado sigue siendo el mismo.



Figuras 33 y 34: Fallos L1 y L2 para experimentos 1 y 2 [ES][Core2Duo]

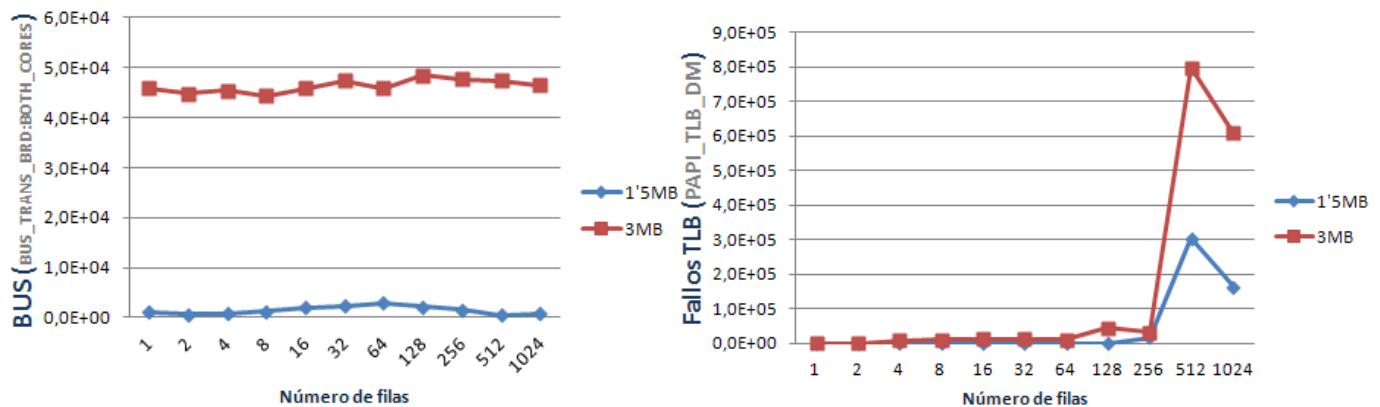
En las anteriores imágenes vemos los fallos en las memorias cachés L1 y L2 respectivamente. Se puede deducir que el cuello de botella que tenemos en los tiempos viene principalmente de las memorias cachés, donde se eleva enormemente el número de fallos. En el caso de la matriz pequeña, que cabe en L2, no tenemos ningún problema y la gráfica se

muestra plana (Figura 34). Si hacemos un pequeño cálculo para ver cuántas filas serían necesarias para conseguir llenar la caché en una iteración tendríamos:

Memoria caché	Nº	Tamaño	Nº filas
Caché L1 Datos	2	32KB	32KB/64B = 512 filas
Caché L2	1	2MB	2MB/64B = 32768 filas

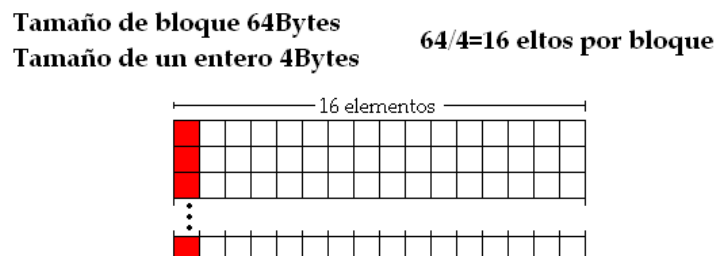
Tabla 6: Número de filas para llenar caché del Core2Duo.

Hemos de tener en cuenta que a pesar de tener dos memorias cachés de datos en este caso sólo estamos empleando un único núcleo, por tanto sólo estamos empleando una única memoria. El número de filas es muy superior al punto en el que comienzan a aumentar las gráficas. Podemos pensar entonces que aunque en la primera iteración no se estén llenando las cachés esto se consigue en las siguientes iteraciones. Sin embargo, el cálculo de la primera iteración es crítico, ya que, si conseguimos llenar completamente la caché, en la siguiente iteración implicará tener que reemplazar el bloque que fue accedido hace más tiempo (LRU) y esto, a su vez repercute en que estaríamos desechando la información que vamos a utilizar a continuación.



Figuras 35 y 36: Transferencias de bus y fallos TLB para experimentos 1 y 2 [ES][Core2Duo]

La ocupación del bus (Figura 35) es bastante superior en el caso de la matriz grande, y es que al no caber completamente en la memoria caché de nivel 2, el microprocesador tiene que estar accediendo constantemente a la memoria principal para recuperar los datos. En cuanto a la Figura 36 que representa los fallos en TLB, se aprecia un salto totalmente brusco cuando comenzamos a contar con 256 filas en nuestro benchmark. Si planteamos lo que está sucediendo al acceder a la primera columna tenemos lo siguiente:



Tamaño de página TLB 4096Bytes

Figura 37: Muestra de utilización en una iteración

Al acceder a la primera columna realmente cargamos los 15 elementos consecutivos a este, luego para saber con cuántas filas tendríamos que utilizar para conseguir un tamaño de una página (4096B) tendríamos que dividir este tamaño entre los 64 Bytes del bloque, y tenemos que con un total de 64 filas tenemos una página exacta. En la figura se observa claramente que cuando intentamos acceder a más del tamaño de una página (256 filas) la TLB comienza a mostrar fallos en su funcionamiento. Con un total de 256 filas estaríamos accediendo a 4 páginas. Esta subida debería producirse teóricamente más tarde, puesto que la TLB tiene una capacidad de hasta 256 entradas para datos <sup>[52]</sup>.

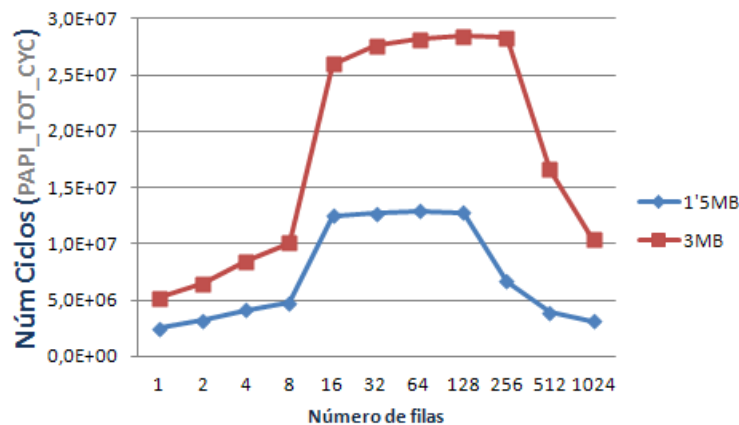


Figura 38: Número de ciclos para experimentos 1 y 2 [ES][Core2Duo]

Los ciclos se ejecutan de forma similar al tiempo aunque estos reflejan un crecimiento ligeramente mayor que en el caso del tiempo para número de filas entre 16 y 256. Vemos directamente que para la matriz pequeña de 64 filas tenemos en torno a 13 millones de ciclos y en el caso de la matriz grande supera el doble, 40 millones de ciclos.

#### 4.4.1.1 Prefetching

En la siguiente experimentación mostraremos el experimento 2, en el cuál cogimos la matriz de 3MB de tamaño. Desactivaremos las opciones de *prefetching* y tendremos cuatro experimentaciones con/sin *prefetching* hardware/software.

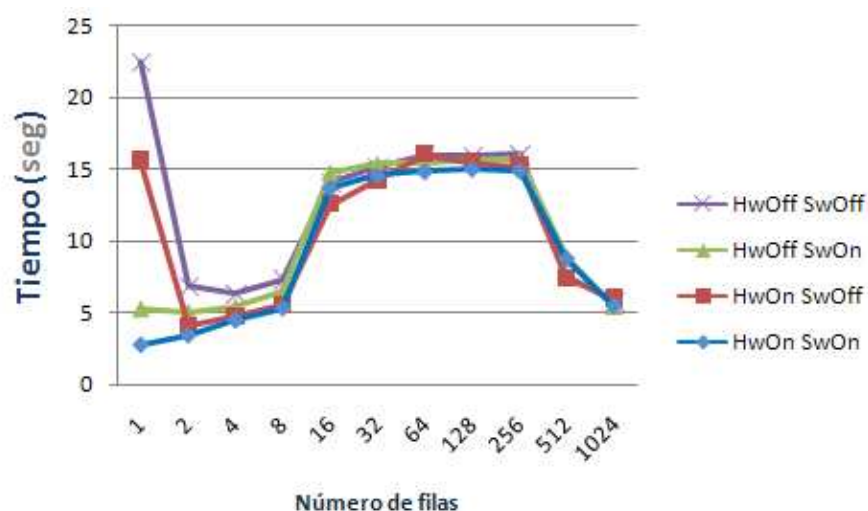
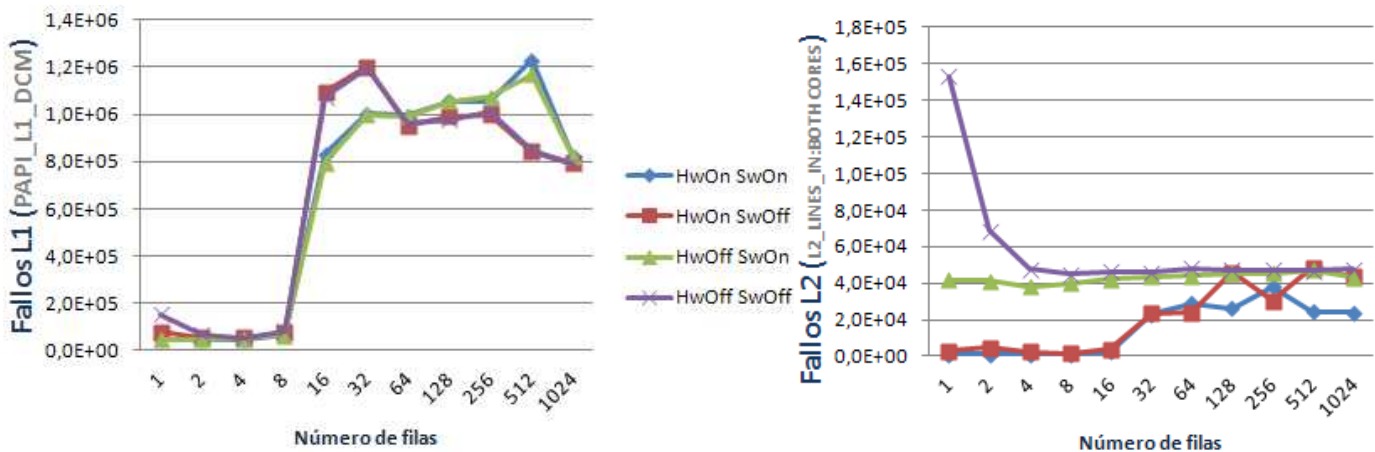


Figura 39: Tiempo registrado para experimentos 2, 3, 4 y 5 [ES][Core2Duo]

En las estimaciones que habíamos hecho sabíamos que al desactivar el *prefetching* los resultados iban a empeorar. Como podemos corroborar la línea azul, en la que todas las opciones están activadas, es la que se mantiene por debajo de todas, en la mayoría de los casos, excepto en algunas ocasiones. De esto podemos deducir que para ciertos números de filas (16, 512), el *prefetching* hardware y software suponen una redundancia. Concretamente es el *prefetching* software el que en estos casos está aumentando el número de ciclos de forma innecesaria. En el resto de valores no hay grandes diferencias y esto se puede deber a que no resulta sencillo predecir el salto, tenemos que tener en cuenta que a pesar de que el salto siempre es constante, cuando llegamos al final del vector realizamos un salto al comienzo del vector, lo que puede descolocar a los algoritmos de predicción.

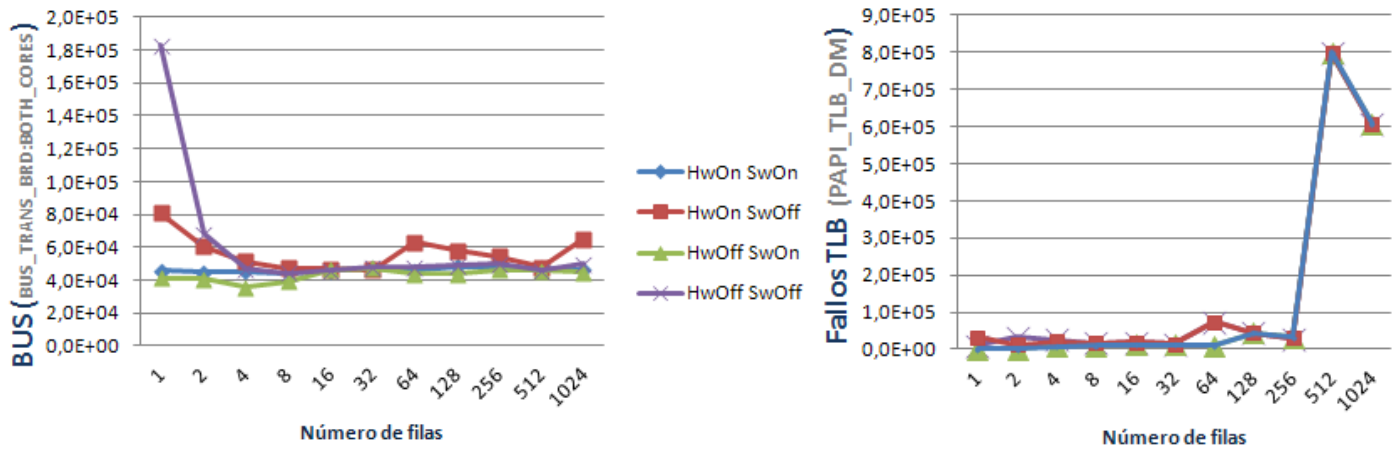
Destaca también sobremanera el gran funcionamiento del *prefetching* para tamaño de grano 1, en este caso si recordamos estamos accediendo al vector de forma consecutiva, y llama la atención como se disparan los tiempos al desactivar el *prefetching* software. Es probable que en estos casos si estamos accediendo al elemento N, el *prefetching* software añada instrucciones de carga del elemento N+1 para tener todos los datos de antemano en la caché.



Figuras 40 y 41: Fallos L1 y L2 para los experimentos 2, 3, 4 y 5 [ES][Core2Duo]

Sobre el funcionamiento de las cachés las diferencias son mayores que respecto a los tiempos. Para tamaños de grano bajos (16 y 32) desactivar el *prefetching* software no resulta beneficioso (véase Figura 40). Como dato curioso, en el resultado con un tamaño de grano 512, obtenemos que desactivar el *prefetching* software puede ahorrar fallos en L1, sin embargo, por los resultados obtenidos en tiempos vemos que este efecto no es de gran importancia.

Desde el punto de vista de fallos en L2 (Figura 41) vemos que resulta crucial tener alguna de las dos opciones de *prefetching* activadas y sobre todo para el caso de acceso consecutivo (número de filas = 1). Como patrón general el *prefetching* hardware parece funcionar mejor ya que, si observamos la comparación de esta opción con aquella en la que tenemos todas las opciones activadas, vemos que las diferencias prácticamente no existen.



Figuras 42 y 43: Transferencias de bus y fallos TLB para experimentos 2, 3, 4 y 5 [ES][Core2Duo]

Si observamos la gráfica de transferencias de bus (Figura 42) deducimos que la desactivación del *prefetching* software supone un mayor problema que la desactivación del *prefetching* hardware. Puede ser que el *prefetching* hardware esté mandando más datos a L2 por precaución de los que son necesarios y esto produzca un mayor número de fallos, sin embargo, cuando ambas opciones no están activadas se reduce el número de transferencias, algo que llama bastante la atención, podría ser que en este caso la técnica de *prefetching* hardware sea capaz de detectar el funcionamiento de instrucciones de carga de tal manera que consiga gestionar de manera más eficiente el número de transferencias de bus. En cuanto al comportamiento de la TLB (Figura 43) no muestra nada nuevo que no viésemos en la anterior prueba.

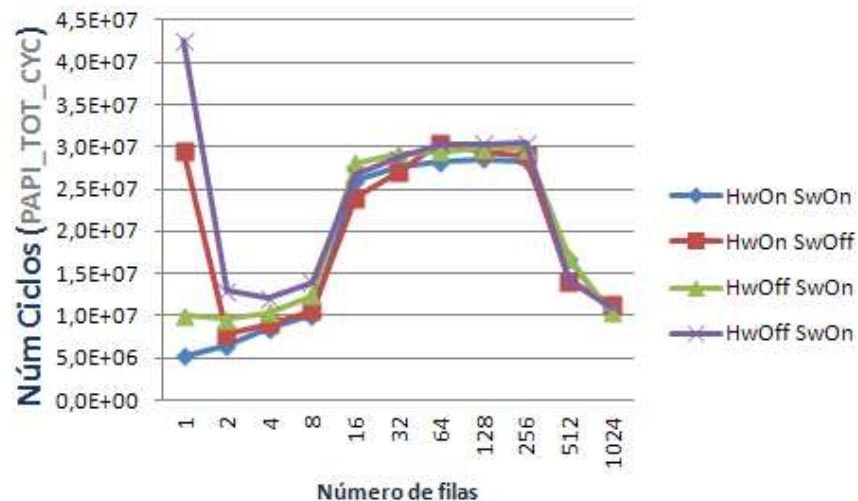


Figura 44: Número de ciclos para experimentos 2, 3, 4 y 5 [ES][Core2Duo]

El número de ciclos refleja que la desactivación de las dos opciones conllevará ejecutar un mayor número de ciclos y, por tanto, supondrá una mayor pérdida de tiempo. Teóricamente el *prefetching* software debería suponer un mayor número de ciclos a la ejecución, y si vemos la serie de color verde vemos como en la mayoría de los casos se encuentra sobre la roja (sólo *prefetching* hardware activado), aunque las diferencias no son abrumadoras. Destaca que en el acceso consecutivo, a pesar de que las dos opciones estén activadas ninguna de ellas provoca redundancias, es decir, que lo que haya cargado una técnica la vuelva a cargar la otra, si esta

última situación se estuviese produciendo, a pesar de tener un mayor número de aciertos en caché, debería suponer un aumento del número de ciclos.

#### 4.4.2 AMD Opteron

Para el AMD Opteron también realizaremos dos pruebas de escritura con stride:

- Matriz de 3MB  $\rightarrow (3 \cdot 1024 \cdot 1024) / 4 = 786432$  elementos. Experimento 6. Esta prueba se realizará para poder comparar esta salida con la del Core 2 Duo.
- Matriz de 7MB  $\rightarrow (7 \cdot 1024 \cdot 1024) / 4 = 1835008$  elementos. Experimento 7. La caché de nivel 2 está formada por 12 memorias privadas de 512 KBytes cada una, lo que hace un total de 6 MBytes. Vamos a superar esta capacidad para ver el comportamiento de las memorias cachés.

La memoria caché de nivel 3 está formada por dos memorias de 6MB, a compartir por dos grupos de 6 núcleos de forma respectiva, pero al ser imposible tomar estadísticas de esta no vamos a intentar superar los 12MB. Como queremos obtener medidas precisas volvemos a ejecutar 15 repeticiones de la misma prueba.

Las salidas obtenidas han sido las siguientes:

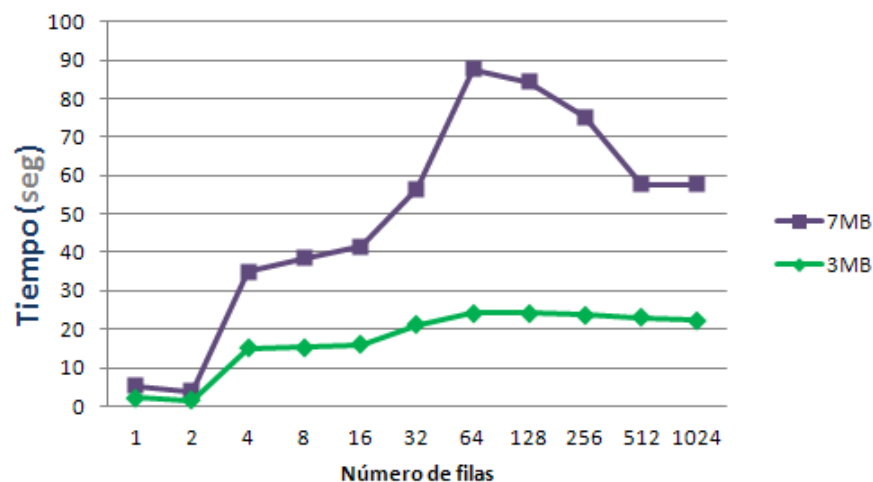
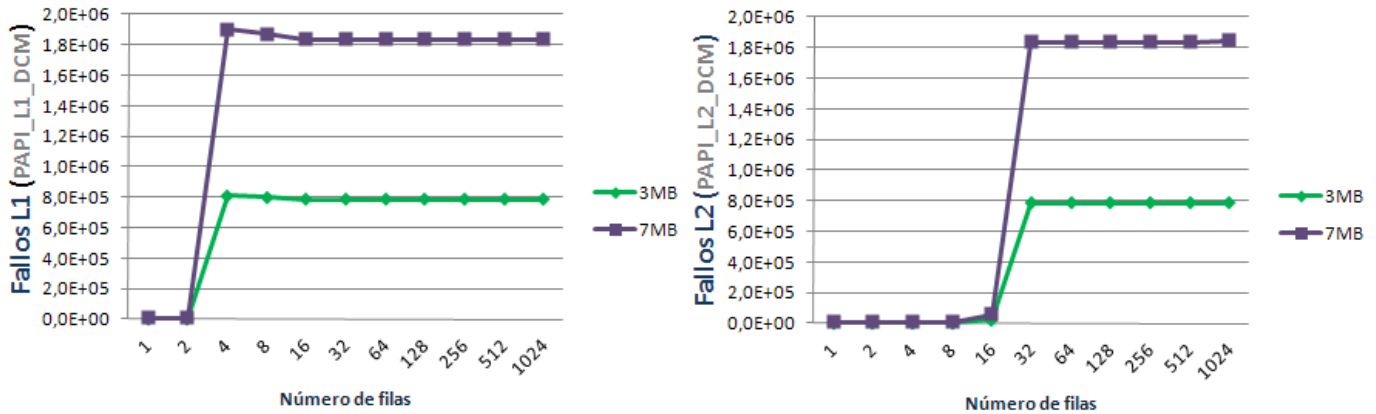


Figura 45: Tiempos para experimentos 6 y 7 [ES][Opteron]

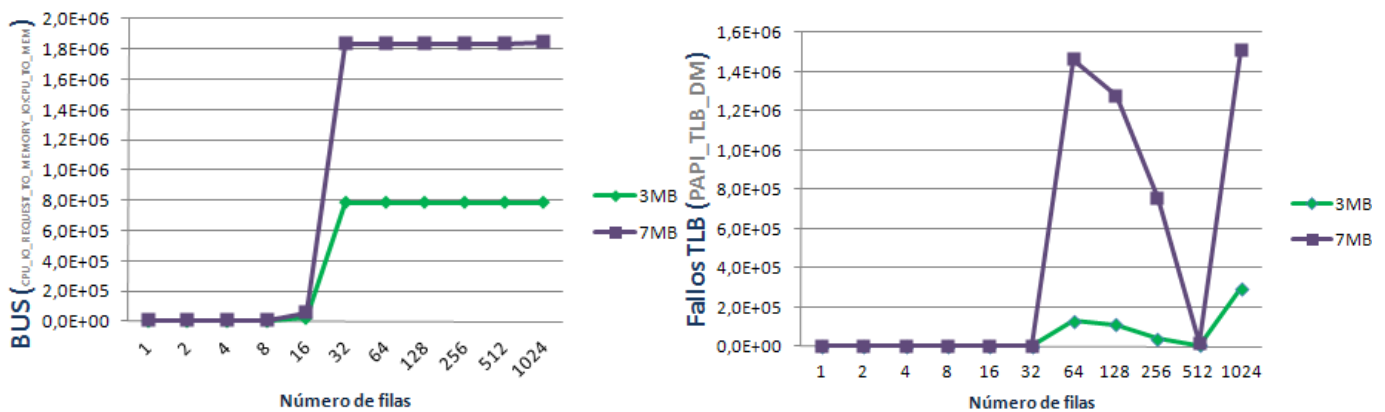
Las pruebas a ser similares a las que lanzamos para el Core 2 Duo, nos muestran resultados similares. La prueba para la matriz de 7MB ofrece muchos más problemas al no caber completamente en las memorias cachés de nivel 1 y 2. Por otra parte el cambio del número de filas no se aprecia prácticamente para la matriz de 3MB. Al igual que sucedía en el Core2Duo los tiempos vuelve a caer; ligeramente para el caso de la matriz más pequeña y de forma mucho más brusca para la matriz de 7 MB.





Figuras 46 y 47: Fallos L1 y L2 para los experimentos 6 y 7 [ES][Opteron]

Los resultados de fallos L1 y L2 son mucho más claros en el Opteron que en el Core2Duo, cuando se llenan ambas cachés pasamos de no tener ningún fallo prácticamente a tener del orden de 1,8 millones de fallos por prueba. Este dato se mantiene constante, algo que también tiene sentido, puesto que la memoria caché se llena completamente y da igual si recorremos la matriz de una forma u otra, al no caber tenemos que cargar nuevos datos, lo que provoca nuevos fallos.



Figuras 48 y 49: Transferencias de bus y fallos TLB para experimentos 6 y 7 [ES][Opteron]

En las transferencias de bus (Figura 48) tenemos un comportamiento similar a los fallos, a partir de cierto número de filas la matriz no cabe en la caché y, por tanto, vamos a realizar más peticiones de datos a la memoria principal.

Sobre los fallos TLB (Figura 49) entendemos que aumenten conforme aumentan el número de filas, debido a que a mayor número de filas más datos necesitaremos y más difícil serán de manejar en las páginas que poseemos; aunque no tenemos explicación para la bajada que se produce para un número de 512 filas con la matriz de 7MB.

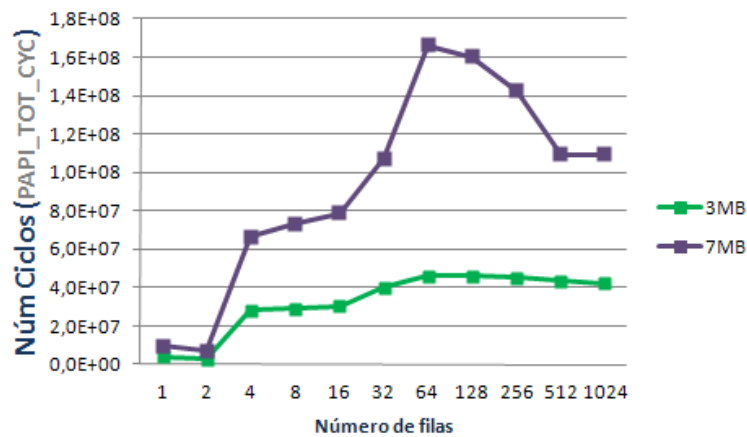


Figura 50: Número de ciclos para experimentos 6 y 7 [ES][Opteron]

Desde el punto de vista de los ciclos vemos un gran aumento provocado por los fallos de caché, aunque vuelve a aparecer el descenso que no somos capaces de predecir.

#### 4.4.3 Comparativa

En este apartado abordaremos la comparativa entre el funcionamiento del Intel Core 2 Duo y el AMD Opteron para los experimentos 2 y 6. En ellos partíamos de una matriz de 3MB, lo que va a hacer que supere en ambos casos las memorias cachés de L2.

El Core2Duo parte con cierta ventaja ya que, la prueba de escritura con stride funciona con un único hilo. Y esto limita las memorias, vamos a ver una tabla de las memorias con las que cuentan.

Característica	Core 2 Duo	Opteron
Procesador (Frecuencia)	1800 Hz	1900 Hz
L1 Datos	32 KB	64 KB
L2	2 MB	512 KB
L3	-	6MB
Transferencia Bus	800 Hz	3200 Hz

Tabla 7: Comparativa de características entre Core2Duo y Opteron.

Los resultados obtenidos fueron los siguientes:

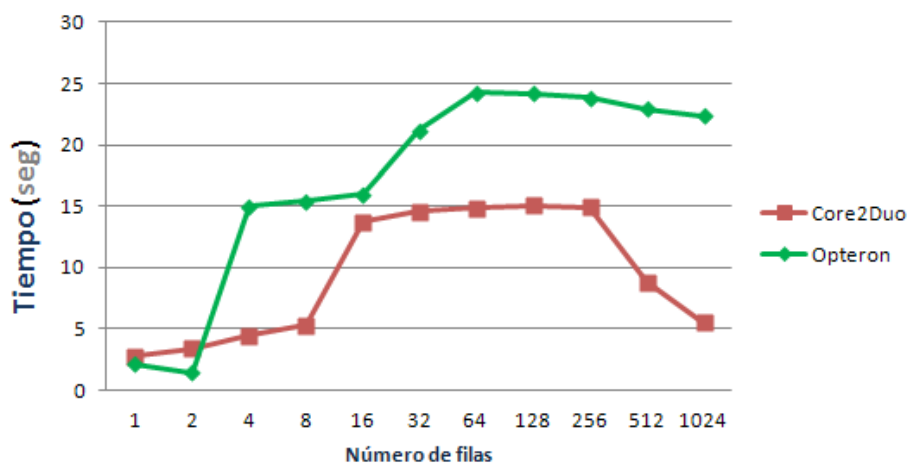
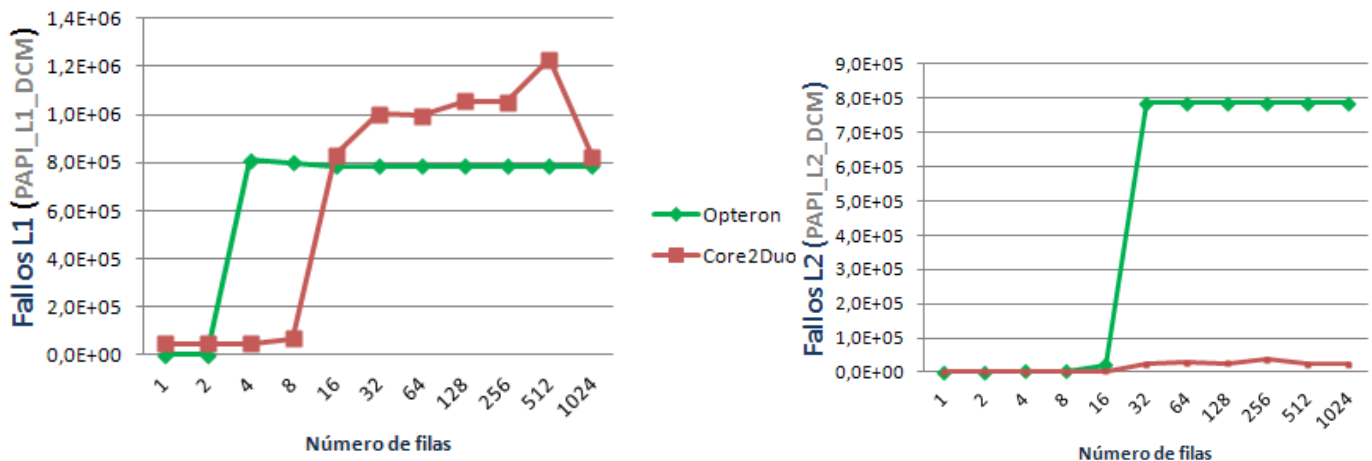


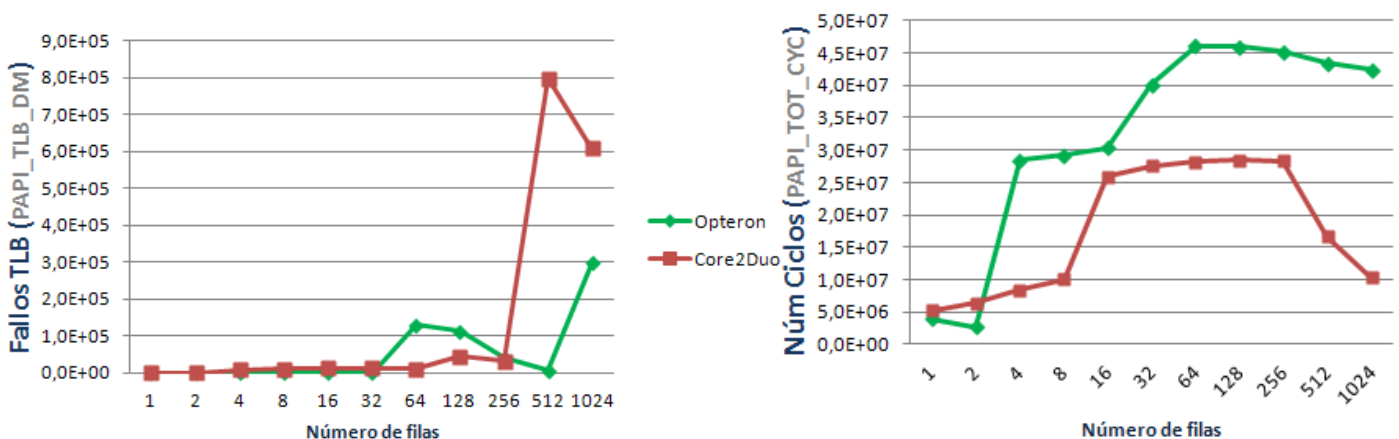
Figura 51: Tiempos para experimentos 2 y 6 [ES][Core2Duo y Opteron]

Sorprendentemente el Core2Duo funciona mejor que el Opteron en la mayoría de los casos, esto vuelve a demostrar lo que dijimos en la introducción de la memoria. En ciertas ocasiones este tipo de arquitecturas como la del Opteron, tiene grandes dificultades para funcionar como aplicaciones monohilo. También es lógico que exista esta diferencia ya que en el Opteron sólo estamos utilizando una doceava parte de su potencial, mientras que en el Core2Duo utilizamos la mitad de sus recursos. Además la diferencia de procesamiento no es muy grande, sólo 100Hz y la ventaja que tenemos con el bus 2400Hz más, no la podemos explotar ya que, aunque la L3 nos va a proporcionar cierta ventaja, el cuello de botella aparece en L2.



Figuras 52 y 53: Fallos L1 y L2 para los experimentos 2 y 6 [ES][Core2Duo y Opteron]

En la Figura 52 se muestra cómo la memoria L1 del Opteron (64KB) se llena antes que la del Core2Duo (32KB), algo que no tiene sentido, pero que repercute de forma directa en los tiempos, haciendo que el Opteron comience a funcionar peor antes. En L2 (Figura 53) podemos decir que se confirman los pronósticos y el Opteron con una L2 mucho más limitada, 512KB frente a los 2MBytes del Core2Duo, eleva su tasa de fallos a ochenta mil fallos por ejecución si tomamos el número de elementos de la matriz (786432), deducimos que aproximadamente por cada diez aciertos cachés tenemos un fallo. En este caso la gráfica de transferencias no la mostraremos por ser prácticamente igual a la de fallos L2.



Figuras 54 y 55: Fallos TLB y número de ciclos para experimentos 2 y 6 [ES][Core2Duo y Opteron]

Para la parte de fallos en TLB y ciclos tenemos gráficas más o menos similares. Podríamos decir a modo de resumen y después de ver las gráficas que, la gran diferencia de estas pruebas reside en los fallos en L1, donde en el Opteron, a pesar de tener una mayor capacidad no es capaz de gestionar de forma correcta la memoria y obtiene un mayor número de fallos en este ámbito.

### 4.5 False sharing

En esta prueba trabajaremos de forma similar a “escritura con stride”. Valoraremos el funcionamiento de forma individual de ambas máquinas, incluyendo un breve estudio sobre *prefetching* en el Core2Duo. En el último apartado veremos una comparativa del funcionamiento de ambas arquitecturas entre sí.

#### 4.5.1 Intel Core2Duo

Las pruebas que vamos a realizar para comprobar el funcionamiento de *false sharing* dentro del microprocesador van a ser las mismas, pero cabe destacar que tendremos un número fijo de filas en este caso 4. Ejecutaremos con un total de dos procesos, ya que esta arquitectura dispone de dos núcleos y realizaremos distintos repartos de carga de forma estática. El reparto se realizará por elementos, y aumentará de forma exponencial grado 2.

Después de realizar la ejecución los resultados son los siguientes:

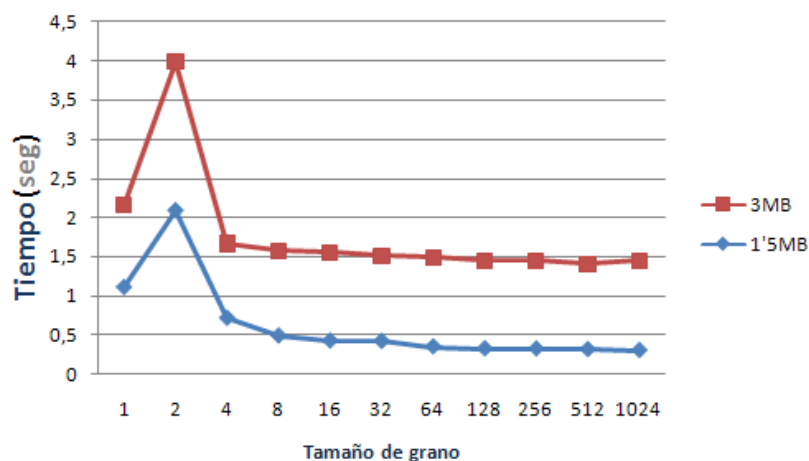
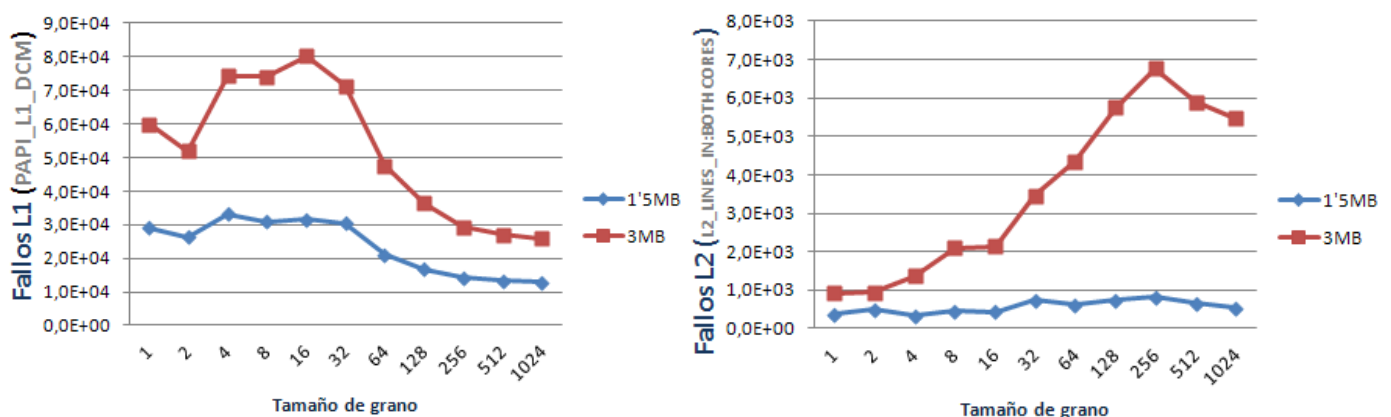


Figura 56: Tiempos para experimentos 8 y 9 [FS][Core2Duo]

En la figura apreciamos cómo hay un descenso generalizado de tiempos con el aumento del tamaño de grano. Con el aumento de éste, conseguimos que ambos procesos compartan un menor número de bloques de memoria. A partir de un tamaño de grano 8 los procesos no escriben en sus respectivas zonas de memoria y pueden funcionar prácticamente de forma independiente.

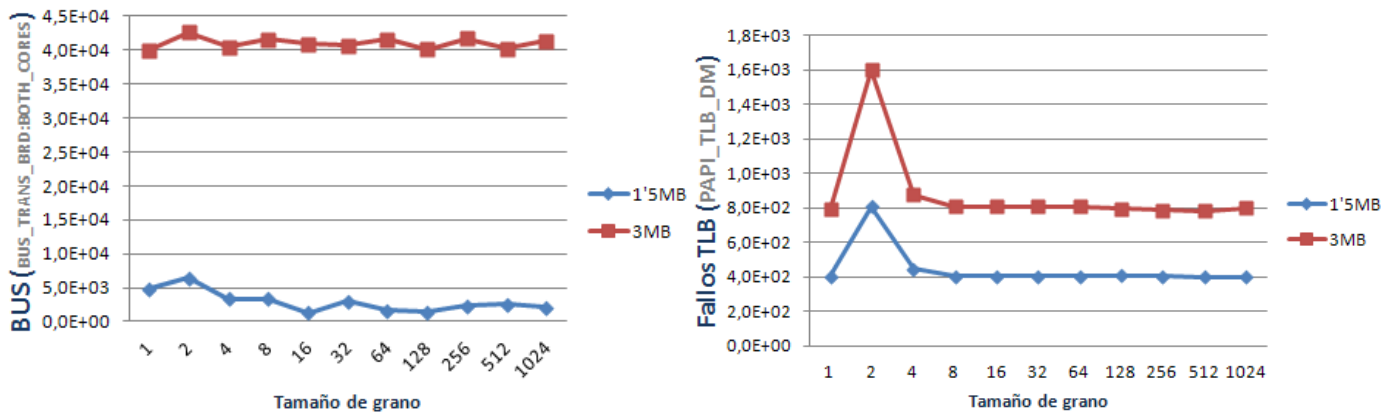


Figuras 57 y 58: Fallos L1 y L2 para experimentos 8 y 9 [FS][Core2Duo]

En cuanto a los fallos de memoria en memoria caché vemos cómo sucede algo similar a la escritura en stride. Como la matriz pequeña entra en memoria no se observa prácticamente ningún error, sin embargo con la matriz grande hay claras diferencias.

Para la caché L1, Figura 57, tenemos el problema de la invalidación, que se produce de forma casi constante para tamaños de grano pequeños. Con el aumento de esta variable vemos cómo se reduce el número de fallos puesto que los bloques comienzan a ser prácticamente dependientes de un único hilo.

En el caso de L2, Figura 58, vemos cómo el caso de *false sharing* no afecta prácticamente, sin embargo, al aumentar el tamaño de grano, y cuando L1 empieza a funcionar correctamente, los fallos se trasladan a L2, esto se produce porque la matriz no cabe en memoria y tenemos que cargar los nuevos datos desplazando zonas de memoria que probablemente utilizaremos en el futuro. El efecto de *false sharing* no se refleja en L2 debido a que es compartida, sin embargo este efecto aparecerá en el Opteron puesto que en su arquitectura cada núcleo tiene una L2 privada.



Figuras 59 y 60: Transferencias de y fallos TLB para experimentos 8 y 9 [FS][Core2Duo]

Las transferencias de bus (Figura 59) se mantienen constantes prácticamente durante toda la ejecución aunque a menor tamaño de grano sí hay más fluctuación. En la gráfica de fallos TLB (Figura 60) observamos que la matriz de mayor número de elementos dobla, en todos los casos el número de fallos, por lo que podemos decir que el número de fallos es directamente proporcional al número de elementos de la matriz.

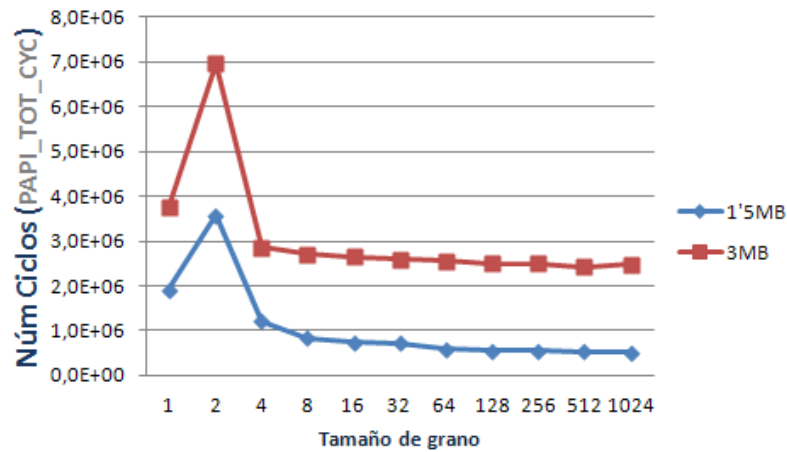


Figura 61: Número de ciclos para experimentos 8 y 9 [FS][Core2Duo]

Para los ciclos sucede algo similar a las anteriores gráficas, para un tamaño de grano pequeño necesitamos un mayor número de ciclos, puesto que las invalidaciones de bloques repercutirán en más instrucciones de carga de datos.

#### 4.5.1.1 Prefetching

Al igual que hicimos un estudio sobre *prefetching* para escritura con stride, haremos nuevas pruebas a partir del experimento 10 deshabilitando las opciones de *prefetching*.

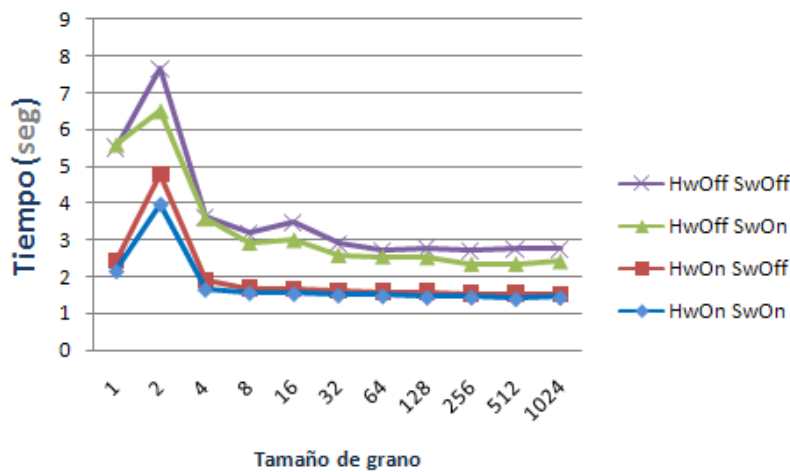
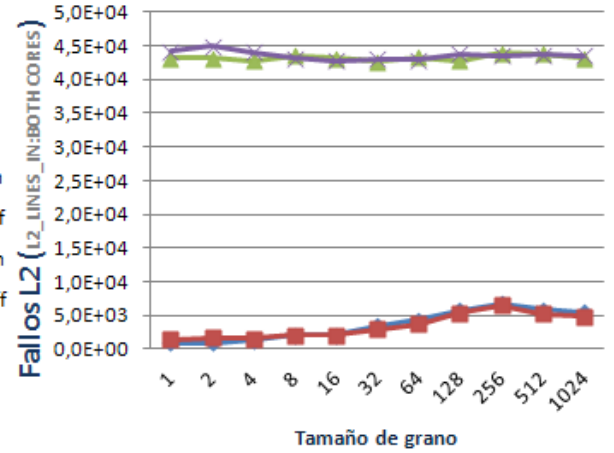
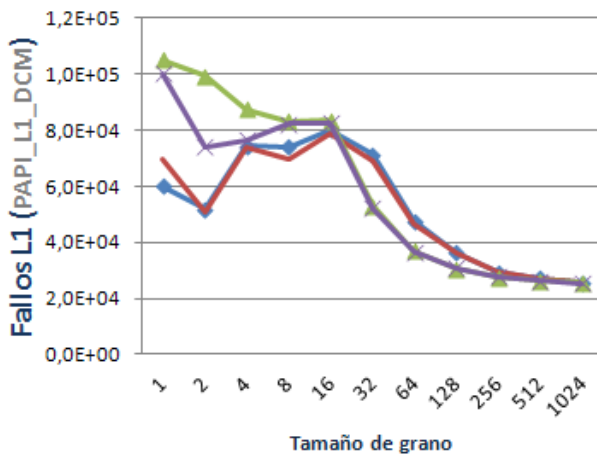


Figura 62: Tiempo registrado para experimentos 9, 10, 11 y 12 [FS][Core2Duo]

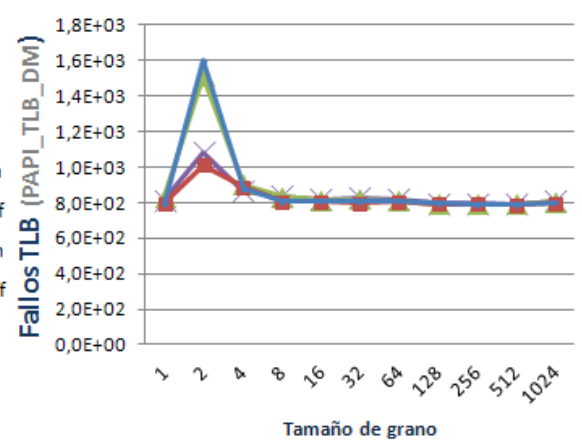
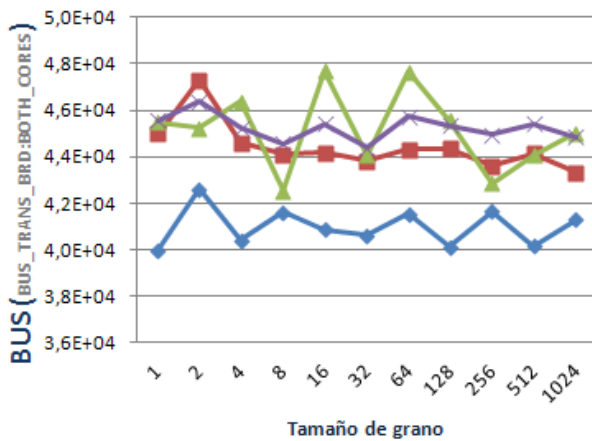
En este caso al tener un acceso consecutivo las diferencias de tiempo son mayores que en el caso de escritura con stride. Esto se debe a que para el *prefetching* es mucho más fácil predecir que si accedemos a la posición N, el siguiente paso será acceder a la posición N+1. De nuevo el *prefetching* hardware parece ofrecer mejores resultados.



Figuras 63 y 64: Fallos L1 y L2 para experimentos 9, 10, 11 y 12 [FS][Core2Duo]

Para los fallos en L1, Figura 63, observamos un descenso mucho más tardío y lento que el que tenemos en tiempos. El efecto de *false sharing* sigue estando presente en el nivel 1 tanto si tenemos algunos bloques precargados o no. El problema de *false sharing* aparece cuando un núcleo modifica zonas de memoria en L1 que utilizará el otro núcleo, esto es totalmente independiente de si el bloque estaba cargado previamente o no.

Desde el punto de vista de L2, Figura 64, tenemos el mismo efecto de subida, aunque en la gráfica prácticamente no se aprecia debido a las grandes diferencias entre la utilización o no del *prefetching* hardware. La desactivación de esta opción hace que aumente el número de fallos en un 900% y un 3600%



Figuras 65 y 66: Transferencias de bus y fallos TLB para experimentos 9, 10, 11 y 12 [FS][Core2Duo]

En la parte de bus, Figura 65, se observa la ganancia de tener las dos técnicas a la vez. Con ellas activadas reducimos de forma importante las transferencias de bus necesarias. De nuevo la TLB, Figura 66, vuelve a probar para un tamaño de grano 2 un gran número de fallos, este número de fallos se ve agravado por la activación del *prefetching* software.



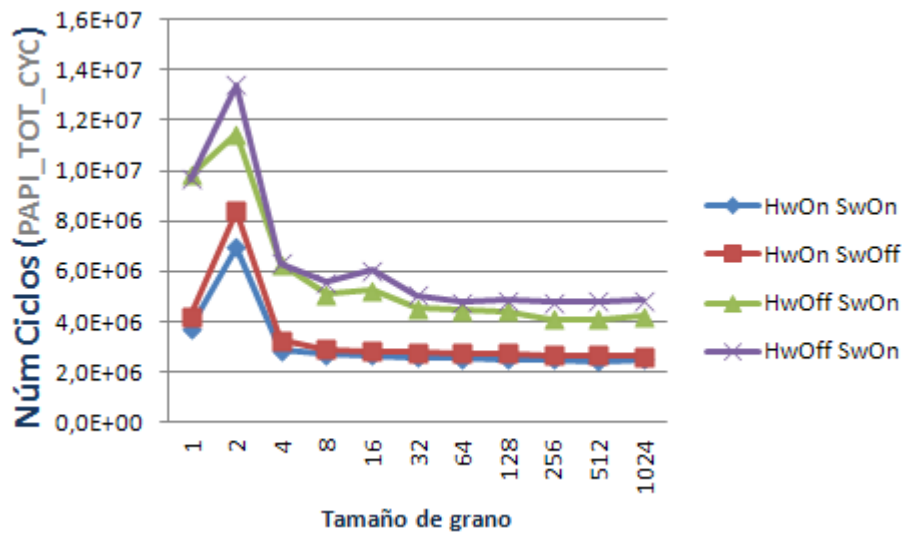


Figura 67: Número de ciclos para experimentos 9, 10, 11 y 12 [FS][Core2Duo]

Los ciclos muestran un poco lo que hemos visto hasta ahora, de nuevo hay un fuerte aumento de ciclos debido a los fallos de TLB en tamaño de grano 2, y desde ahí hacia delante tenemos una bajada de ciclos hasta normalizarse en una recta. En este caso vemos como el *prefetching* software añade un mayor número de ciclos a la ejecución y por ello el funcionamiento en cuanto a tiempos suele ser peor.

#### 4.5.2 AMD Opteron

Las pruebas que ejecutaremos en *false sharing* serán similares, desde el punto de vista del tamaño de la matriz, a las que realizamos con escritura con stride. Una prueba tendrá una matriz de 7MB y otra de 3MB. Esta última será la que volveremos a usar para poder comparar con el funcionamiento del Core2Duo como hicimos en el Apartado 4.3.3.

Las pruebas se realizaron bajo 15 repeticiones y los resultados fueron los siguientes:

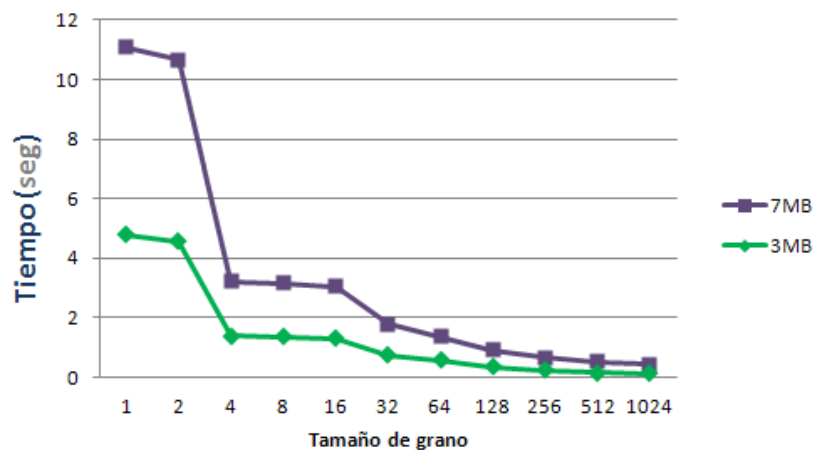
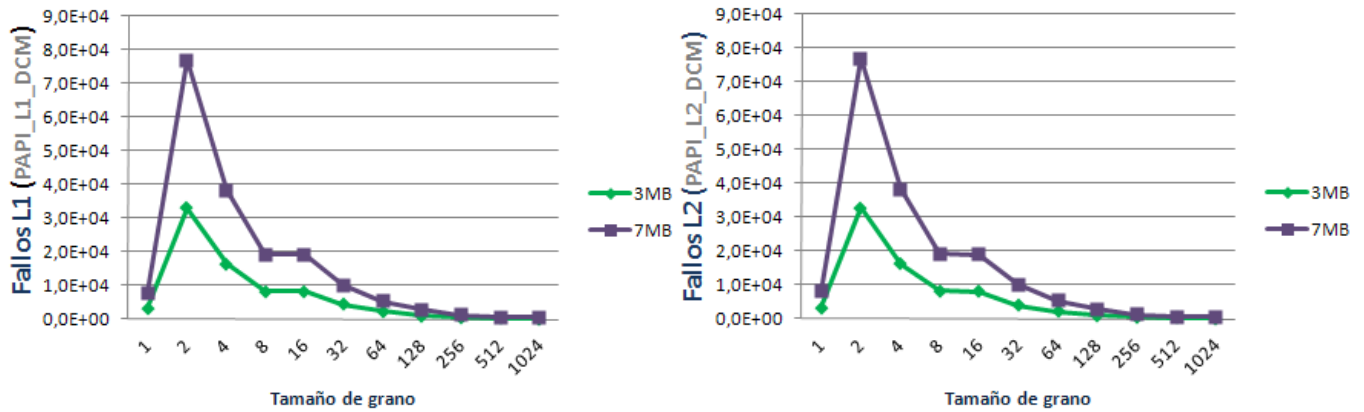


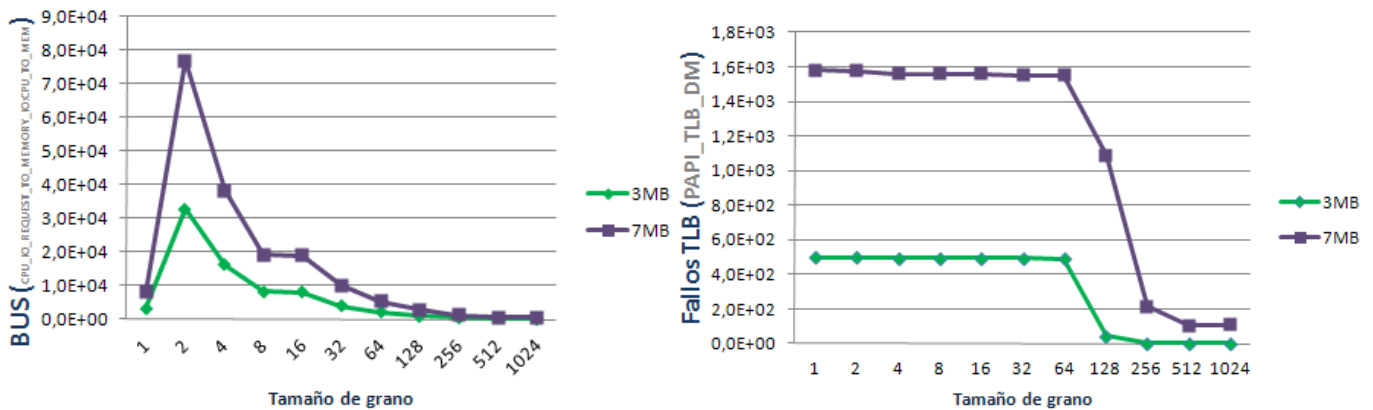
Figura 68: Tiempo registrado para experimentos 13 y 14 [FS][Opteron]

En los tiempos que nos ofrece el Opteron desaparece el comportamiento que teníamos en el Core2Duo (Figura 68), en el cual funcionaba mejor para un tamaño de grano 1 que para un tamaño de grano 2 y los resultados son mucho más razonables. El tamaño de bloque es de 64 Bytes y a partir de los 16 elementos debería notarse un cambio más brusco puesto que todo lo que requiere cada núcleo se encuentra en un bloque. Sin embargo, cuando pasamos de un tamaño de 4 es cuando se produce la principal mejora.



Figuras 69 y 70: Fallos L1 y L2 para experimentos 13 y 14 [FS][Opteron]

Los fallos en L1 y L2 son totalmente idénticos, la explicación para ello es que tenemos una memoria de nivel 1 de 64KB de 2 vías, mientras que la de nivel 2 tiene una capacidad de 512KB y un total de 16 vías. Si hacemos un rápido cálculo  $64KB/2 = 512KB/16 = 32KB$ , nos damos cuenta que la capacidad real de almacenamiento es la misma y que cuando se produce un fallo en la memoria L1, este fallo va a repercutir en otro fallo en L2.



Figuras 71 y 72: Peticiones al bus de memoria y fallos TLB para experimentos 13 y 14 [FS][Opteron]

Las transferencias de bus (Figura 71) reflejan que debido a los fallos de memoria obtenidos en L1 y L2, tenemos un mayor aumento del tráfico del bus para tamaños de grano pequeños. En la TLB, Figura 72, vemos como tenemos un gran número de fallos para un tamaño de grano pequeño y que, en torno a 128, los fallos de TLB vuelven a ser menores. Los fallos son grandes al principio porque una página se encuentra llena de información que están utilizando los otros núcleos, conforme aumentamos el tamaño de grano la página cada vez contendrá más información destinada únicamente para ese núcleo y, por tanto, los fallos cada vez serán menores. Puesto que es capaz de aprovechar mejor el tamaño de una página.

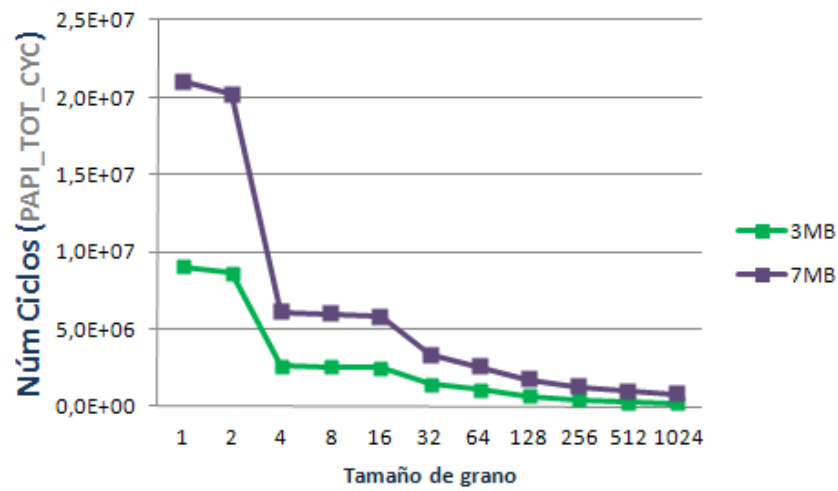


Figura 73: Número de ciclos para experimentos 13 y 14 [FS][Opteron]

Los ciclos no ofrecen nada nuevo, con un menor tamaño de grano requerimos de un mayor número de ciclos para intentar solventar el efecto de *false sharing* que se está produciendo en los tamaños de grano fino.

### 4.5.3 Comparativa

Al igual que en las pruebas de escritura con stride el Core2Duo se veía beneficiado, en este caso será el Opteron el que obtendrá ventajas. Decimos esto porque vamos a ejecutar *false sharing* con el número máximo de núcleos que posea la arquitectura. En este caso la del AMD Opteron se verá beneficiada ya que contará con 10 núcleos más para trabajar en paralelo.

Para realizar la comparativa tomamos la matriz de tamaño 3MB, vemos los resultados a continuación:

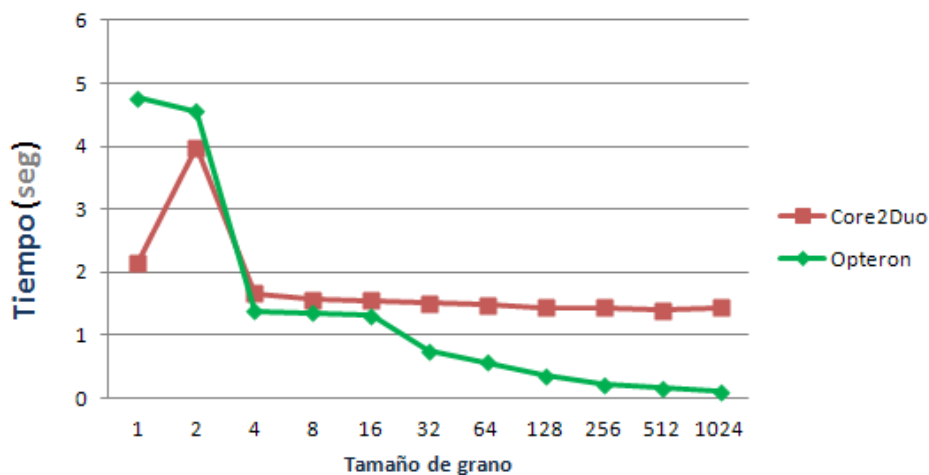
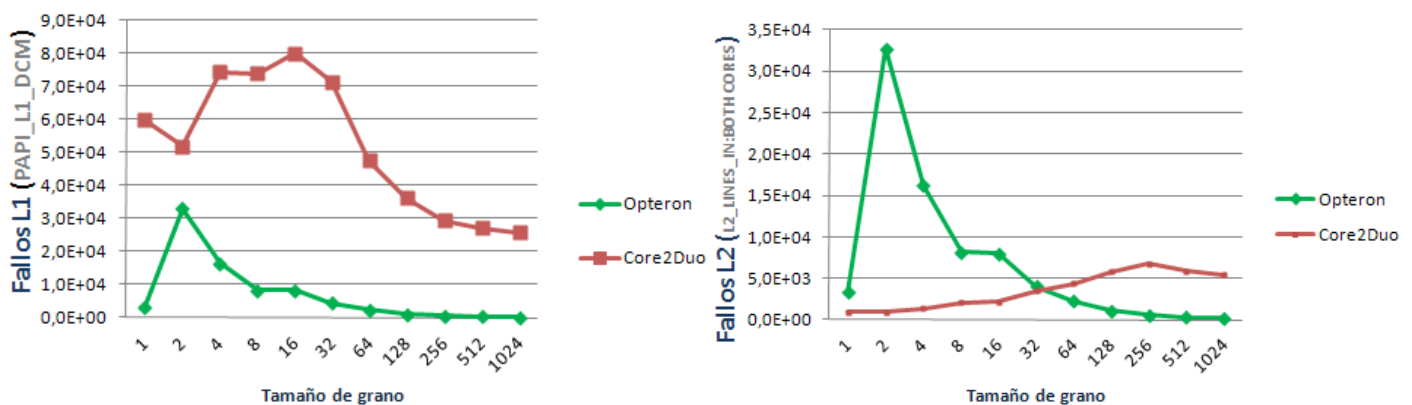


Figura 74: Tiempo registrado para experimentos 9 y 13 [FS][Core2Duo y Opteron]

Los datos corroboran lo que pensábamos que iba a suceder, los tiempos en el Opteron son mucho mejores y consigue ejecutar en un menor número de tiempo las pruebas (véase Figura 74). A pesar de esto tiene mayores dificultades para ejecutar con tamaños de grano pequeño, pero esto tiene una fácil explicación. Si en el Core2Duo tenemos problemas de *false sharing* porque dos procesos escriben en las mismas zonas de memoria, este efecto en el Opteron se multiplica, en el caso de tamaño de grano igual a 1, los doce núcleos del Opteron van a compartir una simple zona de 64 Bytes, en la que todos escribirán.



Figuras 75 y 76: Fallos L1 y L2 para experimentos 9 y 13 [FS][Core2Duo y Opteron]

Sorprendente en L1, Figura 75, los fallos para el Opteron son muy bajos en comparación con los que ofrece el Core2Duo, algo que en un principio no parece tener sentido, sin embargo debemos recordar que PAPI nos ofrece las estadísticas de un solo hilo. Por ello, en este tipo de comparaciones deberemos ver si los datos ofrecidos por el Opteron mejoran 6 veces los datos

del Intel (Core2Duo nos muestra información de 1/2 núcleos y Opteron 1/12). Por tanto, puntualizada esta diferencia podemos explicar que los fallos serían de la siguiente forma en L1:

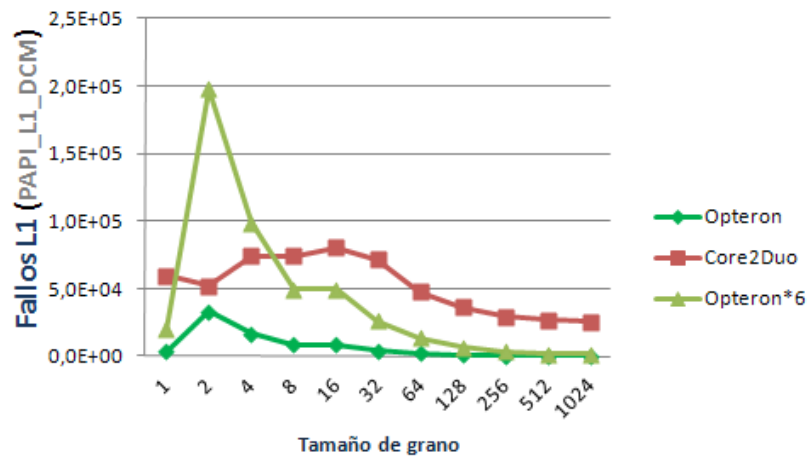
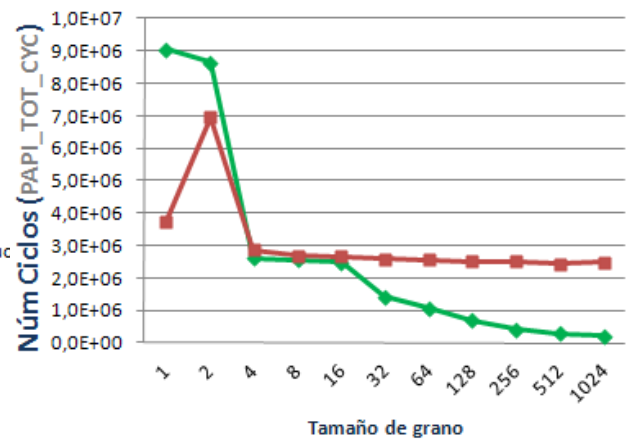
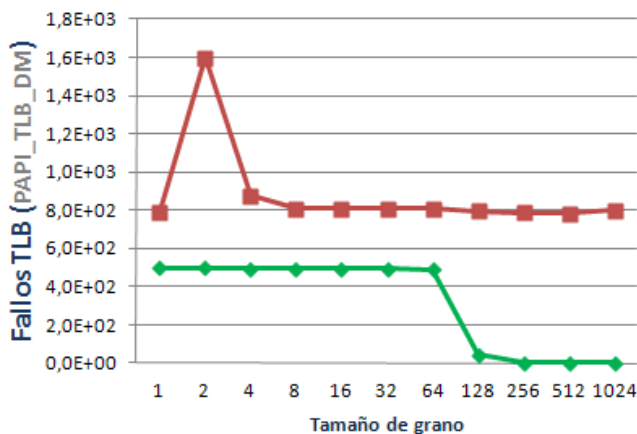


Figura 77: Explicación comparativa fallos L1 en false sharing [FS][Core2Duo y Opteron]

En esta nueva gráfica (Figura 77) se acentúa el efecto de *false sharing*, y veríamos como con 12 hilos el número de fallos es mayor, sin embargo, cuando consigue un tamaño de grano razonable los resultados son los mismos. Lo que quiere decir que tiene el mismo número de fallos pero ejecutando 6 veces más rápido.



Figuras 78 y 79: Fallos TLB y número de ciclos para experimentos 9 y 13 [FS][Core2Duo y Opteron]

En los fallos TLB (Figura 78) tenemos el mismo efecto que estábamos teniendo con escritura en stride, los fallos son mayores con menor tamaño de grano porque las páginas comparten zonas de memoria de distintos procesos, lo que hace poco eficiente el funcionamiento de la TLB. Conforme aumenta este tamaño de grano tenemos cada vez más datos que serán solamente utilizados por ciertos núcleos.

En cuanto a los ciclos (Figura 79) dan una visión aproximada a los tiempos aunque recordamos que estos datos se ofrecen para un solo hilo, por tanto, el Opteron ejecutará más ciclos, pero al ejecutarlos en paralelo el tiempo no se verá afectado por ello.

## 4.6 Producto Matriz-dispersa vector

El tercer gran grupo de pruebas trata sobre el producto entre una matriz dispersa y un vector. Dentro de este tipo de pruebas dividiremos los test en cuatro partes principales en las que trataremos con matrices de distintos grados de dispersión, matrices con distintos tipos de estructuras, variaremos el número de hilos con el que ejecutan las máquinas y, por último, modificaremos las opciones de planificación de OpenMP.

Pero antes de ver estos distintos tipos de propuestas vamos a realizar un pequeño análisis previo de los factores que afectan a esta función. Sabemos que la función consiste en una multiplicación entre una matriz y un vector, pero... ¿cómo afectan estos elementos al resultado final? Para ello hemos diseñado dos pruebas a partir del producto matriz-dispersa vector por filas. En la siguiente figura mostramos el original y posteriormente veremos los dos casos.

En los códigos que vemos “m” representa el número de filas de la matriz, en “Ai” tenemos el vector de punteros a fila que nos dice por cada posición cuantos elementos hay en dicha fila. “Ap” contiene la columna en la que se encuentra situado el valor. Ax representa los valores reales de la matriz y los vectores “x” e “y” representan el vector por el cual se multiplica y el vector de suma, en este último también se almacenan los resultados.

```
#pragma omp parallel for shared(y,Ai,Ax,Ap,x,m) private (p,i)
schedule(SCHEDULE, chunk)
for (i = 0 ; i < m ; i++){
    for (p = Ai [i] ; p < Ai [i+1] ; p++){
        y [i] += Ax [p] * x [Ap[p]];
    }
}
```

Código 7: Producto matriz-dispersa vector, versión creada por nosotros para que se realice por filas

- La primera variación consiste en un producto en el que cada hilo almacenará sus resultados en una variable auxiliar y al final se aplicará el operador *reduction* de OpenMP haciendo una suma de todas estas variables auxiliares. El resultado sigue siendo el mismo puesto que en la versión original devolvemos la suma de los elementos del vector. En este caso veremos cómo afecta la utilización de un vector de forma compartida por distintos hilos pasando a tener variables independientes privadas. En el siguiente recuadro vemos el código.

```
#pragma omp parallel for reduction(+:yaux) shared(y,Ai,Ax,Ap,x,m)
private (p,i) schedule(SCHEDULE, chunk)
for (i = 0 ; i < m ; i++){
    for (p = Ai [i] ; p < Ai [i+1] ; p++){
        yaux += Ax [p] * x [Ap[p]];
    }
}
```

Código 8: Producto matriz-dispersa vector, sin utilizar compartición de vector escritura

- El otro caso consiste en prescindir de la matriz dispersa (Ax en el código) y cargar los valores del vector de punteros fila. De esta forma vemos el impacto de la matriz en el resultado final.

```
#pragma omp parallel for shared(y,Ai,Ax,Ap,x,m) private (p,i)
schedule(SCHEDULE, chunk)
for (i = 0 ; i < m ; i++){
    for (p = Ai [i] ; p < Ai [i+1] ; p++){
        y [i] += Ap [p] * x [Ap[p]];
    }
}
```

Código 9: Producto matriz-dispersa vector, sin utilizar la matriz dispersa (se usa un vector)

Después de implementar estas partes los resultados son los siguientes:

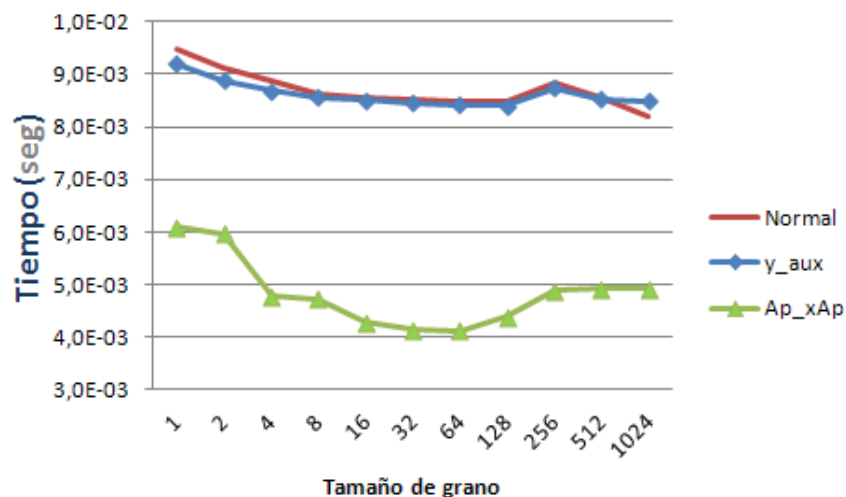


Figura 80: Tiempos para el experimento 15' [PMV][Opteron]

La Figura 80 refleja claramente cómo el peso fundamental de la función lo tenemos en la carga de la matriz, esta matriz tiene tantos elementos que su carga supone más del doble de los fallos que se producirán en la caché. Por su parte el vector afecta en menor escala, y se podría decir que en los tamaños de grano más finos, hay un efecto ligero de *false sharing* que distancia a ambas gráficas hasta que el tamaño de grano es 8.

#### 4.6.1 Dispersión

Dentro de este grupo de pruebas trabajaremos con matrices que tendrán el mismo número de elementos, pero tendrán diferentes dispersiones. Esto quiere decir que los elementos dentro de la matriz se encontrarán localizados de una forma más o menos próxima dependiendo del ancho de banda elegido. Para trabajar con un conjunto cerrado de dispersiones hemos seleccionado 5 tipos de dispersión de: 1%, 25%, 50%, 75% y 100%.

Otros parámetros posibles que permiten variar el generador de matrices son: el número de filas que tendrá la matriz y el número de elementos por fila. Con el primero definiremos el número de filas y columnas que poseerá la matriz, siempre tratamos con matrices cuadradas de dimensión  $m \times m$ . Con el segundo podremos determinar el número de elementos totales de la matriz. Éste último no puede ser muy grande ( $>1000$ ), ya que en caso de superar esta cifra las pruebas con el Core2Duo llevarían demasiado tiempo de ejecución.

Por tanto, los valores seleccionados son 10000 filas por matriz y 500 elementos por fila. En la Figura 20 que vimos en la explicación del producto matriz-dispersa vector, se muestra el patrón de las matrices de prueba, pero con 5000 filas.

Contrariamente a lo que podemos pensar, la dispersión no afectará directamente al acceso que realizamos a la matriz. Si realizamos la comparación de una matriz dispersa y una matriz más concentrada la representación en tres vectores quedaría de la siguiente forma:

- Un vector para representar los valores que tendría tamaño igual al número de elementos de la matriz, y que en ambos casos sería el mismo.
- El vector columnas, que tendría tantas entradas como elementos de la matriz, en este caso la única diferencia sería que los valores de ambos vectores serían bastante dispares.
- El vector de punteros, de tamaño igual a número de filas + 1. El generador de matrices que poseemos crea matrices que intentan mantener constante el número de elementos por fila, por tanto, este vector será muy parecido en ambos casos de matriz concentrada y dispersa.

Sin embargo, la diferencia principal y substancial la vamos a encontrar a la hora de acceder al vector por el que multiplicamos la matriz. Los accesos en los casos de una matriz concentrada serán más contiguos, mientras que en el caso de una matriz dispersa serán mucho más extensos.

En la Figura 81 vamos a mostrar la diferencia en cuantos accesos de forma gráfica:

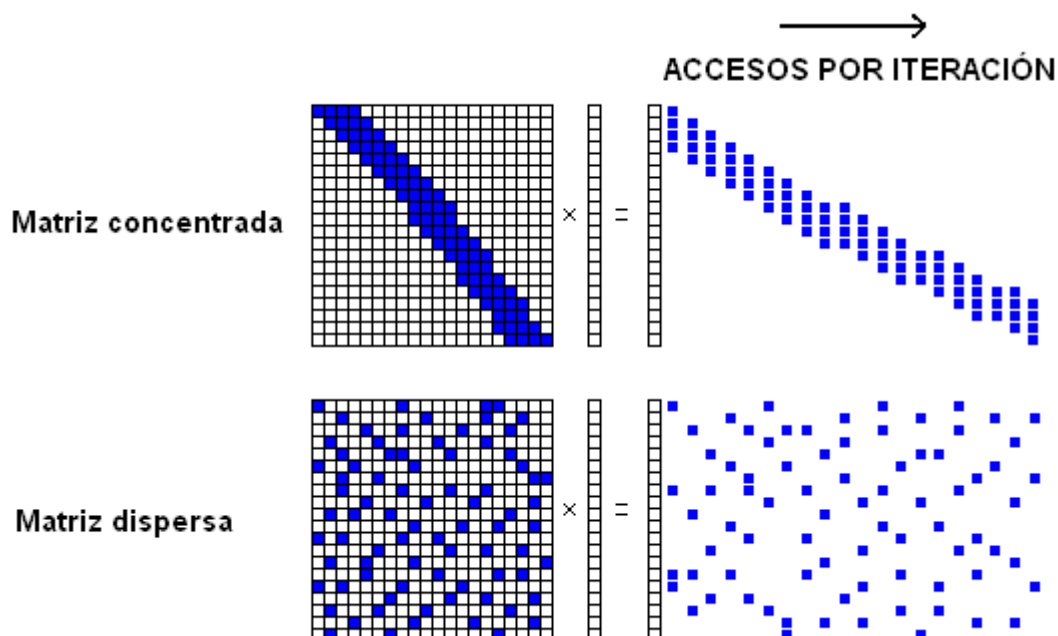


Figura 81: Accesos iteración a matriz sus elementos concentrados o disgregados

En el caso de multiplicar por la matriz concentrada hay mayor probabilidad de que los valores que necesitamos utilizar ya hayan sido precargados anteriormente para anteriores operaciones. En el caso de la matriz dispersa cada vez estaremos cargando más bloques, y la



política de reemplazo expulsará inevitablemente bloques de memoria que utilizaremos en la siguiente iteración.

En cuanto al tamaño de estos vectores en todos los casos vamos a superar el tamaño de las memorias cachés. Vamos a calcular cuánto volumen de datos tendremos. La fórmula del producto viene dada por:

$$y = (T * x) + y$$

Donde “T” es la matriz dispersa, “x” es el vector por el que multiplicamos e “y” es la matriz de escritura donde se almacena el resultado, en un principio está inicializada a “0”. Y por ello, su suma no afectará al resultado final.

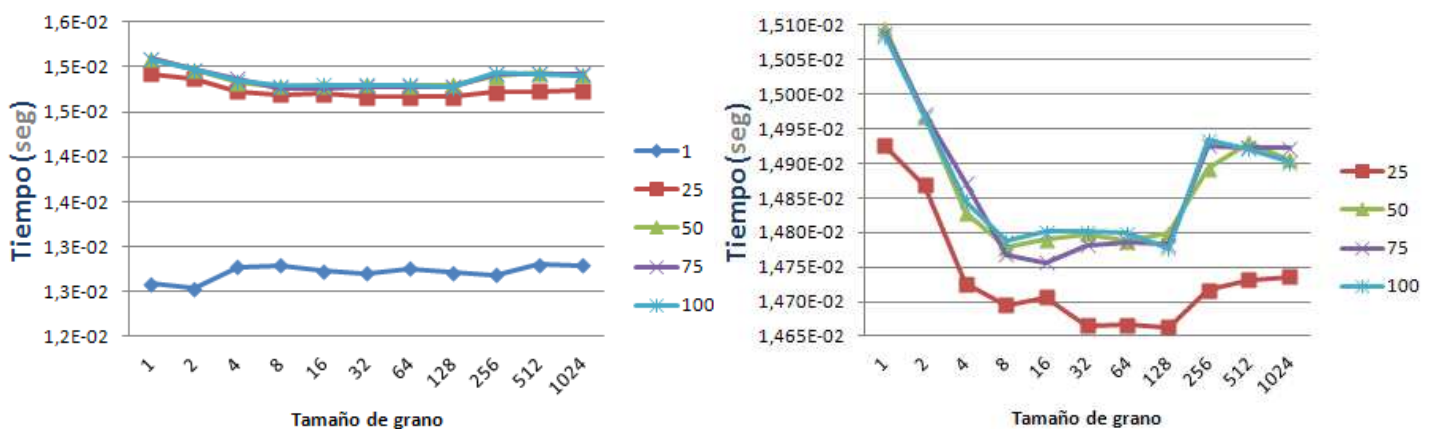
Fórmula	Vector	NNZ	Tipo (Bytes)	Total (MBytes)
<b>T</b> (Matriz dispersa)	<b>Valores</b>	$5,0 * 10^6$	double (8B)	38,14 MBytes
	<b>Columnas</b>	$5,0 * 10^6$	int (4B)	19,07 MBytes
	<b>Puntero a filas</b>	$10^5 + 1$	int (4B)	0,38 MBytes
<b>x</b>	<b>Vector multiplicación</b>	$10^5$	double (8B)	0,72 MBytes
<b>y</b>	<b>Vector resultados</b>	$10^5$	double (8B)	0,72 MBytes
<b>Total</b>				<b>59,03 MBytes</b>

Tabla 8: Cálculo de tamaño de vectores en producto matriz-vector.

El mayor tamaño estará por tanto concentrado en la matriz dispersa. Sería ideal para la CPU conseguir mantener los vectores “x” e “y” íntegros en las memoria caché, ya que son información que va a ser utilizada constantemente, sin embargo, va a ser prácticamente imposible ya que, superan el tamaño de las cachés de nivel 1.

#### 4.6.1.1 Intel Core2Duo

A continuación mostramos las salidas del experimento 15, recordamos que se trata de un producto matriz-dispersa vector sobre matrices con distintos grados de dispersión:

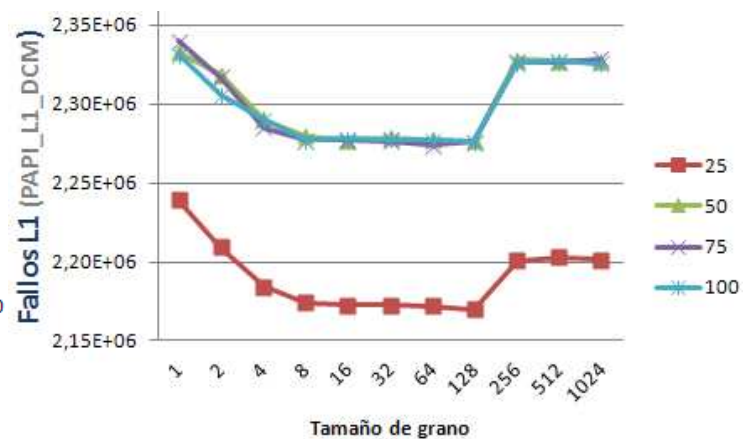
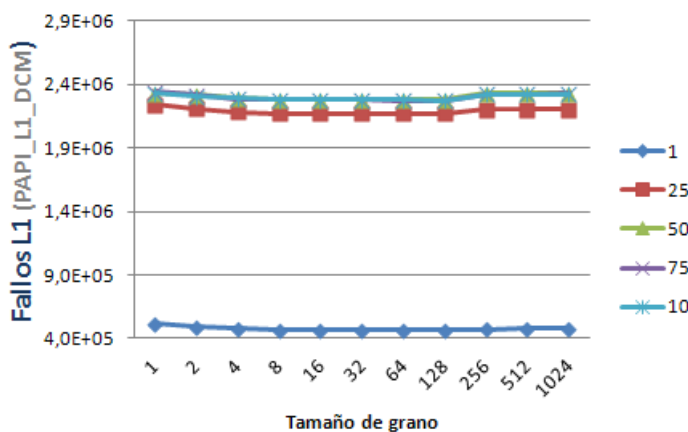


Figuras 82 y 83: Tiempos para el experimento 15 con/sin dispersión 1 [PMV][Core2Duo]

En la Figura 82 se demuestra que para una dispersión menor el funcionamiento del Core2Duo es mejor. Hay una gran diferencia entre una dispersión de 1% y una dispersión del 25%. Esto se debe a que en el primer caso tenemos una matriz prácticamente diagonal, muy concentrada, que accede al vector por el cual se multiplica de forma consecutiva. Si quitamos los datos de la gráfica del 1%, Figura 83, también vemos cosas curiosas. Cuando tenemos un tamaño de grano bajo los núcleos funcionan bastante mal, en este caso también se está produciendo *false sharing* en el vector de escritura.

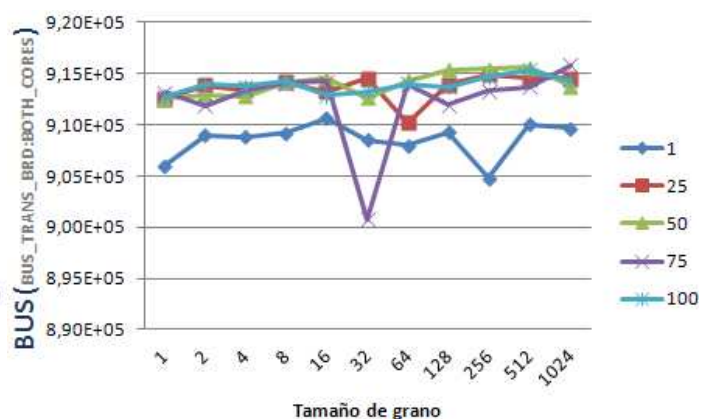
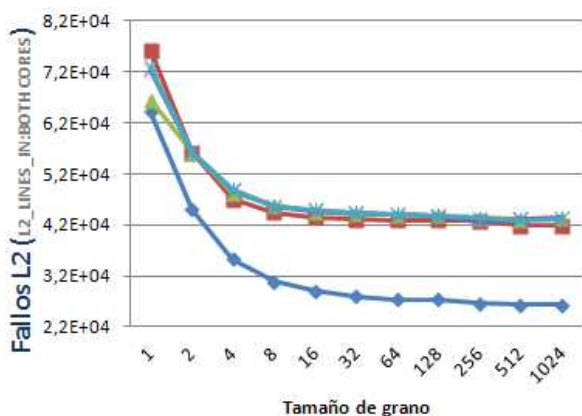
Aunque a cada núcleo le estamos designando una única fila con 500 elementos, tenemos que saber que a la hora de escribir en el vector de resultados (y), si tenemos un tamaño de grano igual a 1, un núcleo estará escribiendo en las posiciones pares y otro en las posiciones impares, por lo que compartirán prácticamente toda la memoria de escritura posible.

En cuanto a la subida de tiempos (Figura 83) puede deberse a una mala planificación, el vector de multiplicación cada vez va siendo más grande y las operaciones por tanto cada vez son más costosas. Para ver si de verdad está sucediendo algo así en el Apartado 4.5.4 veremos lo que sucede con los distintos tipos de planificación. Además la gráfica de la derecha puede ser interesante de cara a buscar un mínimo mediante alguna de las técnicas de optimización que veremos en el Apartado 4.8.



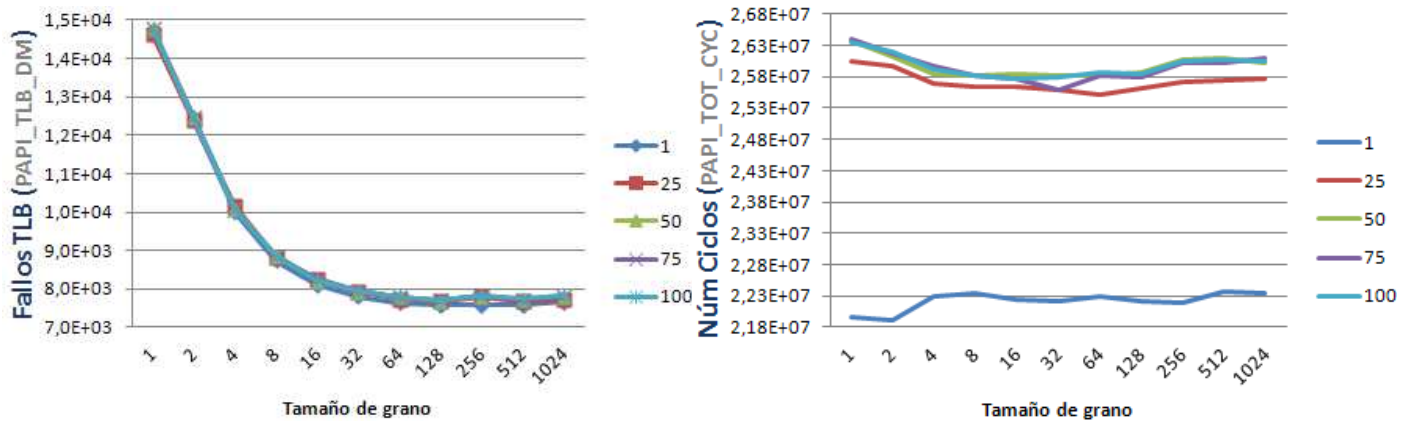
Figuras 84 y 85: Fallos L1 para el experimento 15 con/sin dispersión 1 [PMV][Core2Duo]

Las gráficas de L1, muestran algo similar a lo que hemos visto en tiempos, la gran diferencia de tiempos se debe principalmente a la diferencia de fallos en L1 entre las distintas dispersiones. Si eliminamos de nuevo la gráfica de dispersión 1% (Figura 85) volvemos a ver gráficas en forma de U.



Figuras 86 y 87: Fallos en L2 y transferencias de bus para el experimento 15 [PMV][Core2Duo]

Los fallos en L2 (Figura 86) nos dicen que a menor tamaño de grano más problemas tenemos. A pesar de tener una L2 compartida ambos procesos están actualizando constantemente y esta información debe subir hasta L2, para que ambos núcleos puedan comunicarse. En la parte de transferencias se observa que para la menor dispersión posible el tráfico es menor y para el resto de caso es bastante similar. En el caso de dispersión 1% es probable que los vectores de multiplicación y escritura estén contenidos en L2 constantemente.



Figuras 88 y 89: Fallos en TLB y número de ciclos para el experimento 15 [PMV][Core2Duo]

Para el número de fallos en TLB, Figura 88, se dispara para tamaños de grano pequeños, esto se puede deber a que cuando los procesos acceden a una fila (tamaño de grano), dentro de los 4096 Bytes que representa una página, ambos procesos están compartiendo zonas de memoria que van a ser cargados, sin embargo cuando va creciendo el tamaño de grano cada vez las páginas son más independientes y acaban conteniendo la información concerniente a un único núcleo.

Desde el punto de vista del número de ciclos, Figura 89, vemos algo parecido a los tiempos. Hay una gran diferencia entre la matriz de dispersión 1% y el resto, provocado como hemos dicho por los fallos en la caché de nivel 1.

### 4.6.1.2 AMD Opteron

Realizamos las pruebas para el Opteron con las matrices del mismo tamaño y grado de dispersión.

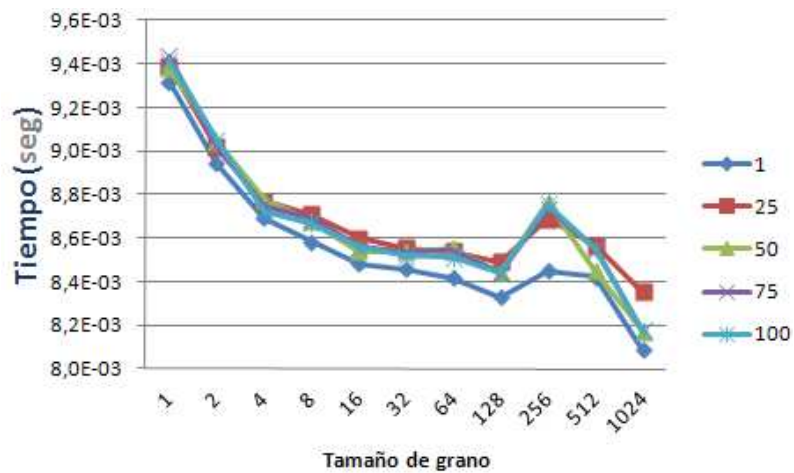
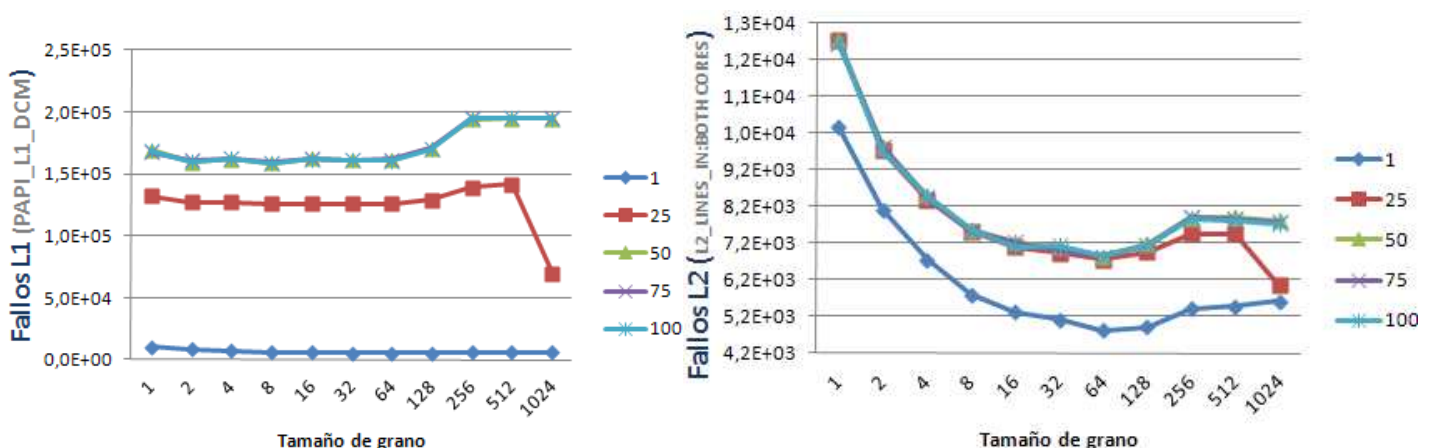


Figura 90: Tiempo para el experimentos 16 [PMV][Opteron]

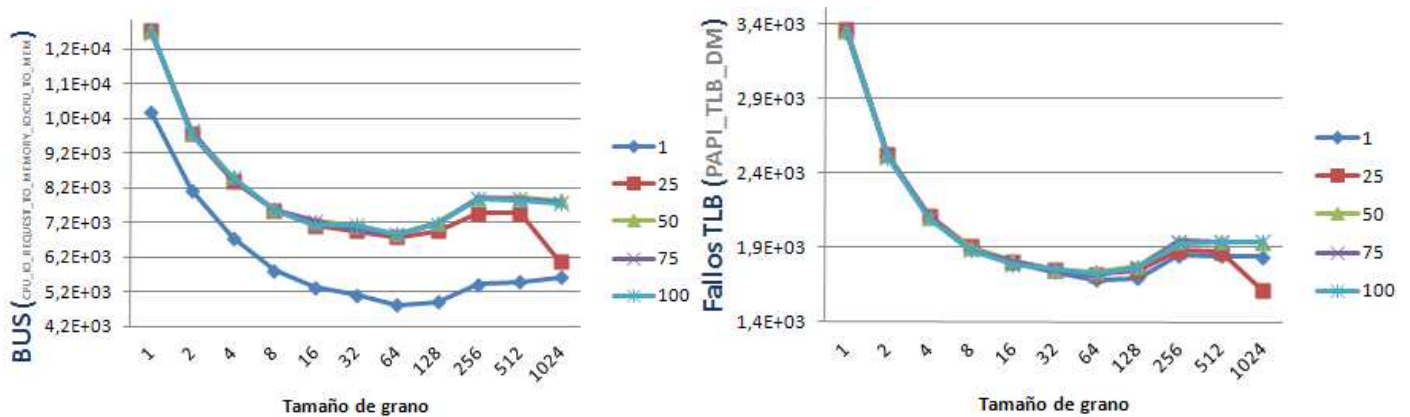
El comportamiento en este caso no es exacto al del Core2Duo, en el anterior caso teníamos una gráfica en forma de U, sin embargo en este caso aumentando el tamaño de grano seguimos teniendo mejoras. Esto no tiene mucho sentido desde el punto de vista teórico, puesto que si tenemos un total de 10.000 filas y estamos utilizando un tamaño de grano de 1024 al hacer la división ( $10000/1024$ ) tenemos que podemos repartir la carga entre "9,76 núcleos" ya que a cada núcleo se le repartirán las siguientes 1024 filas disponibles. Por consiguiente, alguno de los núcleos estará ocioso, con el supuesto aumento de tiempos, que no se está produciendo. Si nos fijamos ahora en el funcionamiento con distintos tipos de dispersión parece que los mejores tiempos se presentan la matriz de menor dispersión, algo que tiene sentido puesto que estamos accediendo a valores próximos en el vector de multiplicación. Sin embargo la diferencia entre el resto de casos no es tan grande.



Figuras 91 y 92: Fallos en L1 y L2 para el experimento 16 [PMV][Opteron]

Los fallos tienen un comportamiento similar en todos los casos, excepto en el caso de dispersión 1 (Figuras 91 y 92), en este caso es probable que el Opteron consiga almacenar el vector de multiplicación en L3. Los fallos en memoria caché no son capaces de darnos visión

del porqué de la caída de tiempos tan fuerte con un tamaño de grano 1024, aunque sí se vislumbra una fuerte bajada para una dispersión del 25%.



Figuras 93 y 94: Peticiones al bus de memoria y fallos TLB para el experimento 16 [PMV][Opteron]

Las transferencias de bus, Figura 93, nos muestran un poco el reflejo de los fallos en L2, cuantos más fallos tenemos en L2, más veces consultaremos en L3 y cuando este dato no esté en L3, usaremos el bus para obtener la información. Si comparamos los valores de la gráfica de L2 y peticiones al bus de memoria no hay gran diferencia; esto nos dice que la L3 en este caso no está aportando gran ayuda. En cierta parte tiene sentido, ya que constantemente vamos a estar cargando valores de la matriz que ocupa aproximadamente 59 MBytes y por tanto los 12 MBytes de la L3 se nos quedan cortos.

#### 4.6.1.3 Comparativa

Para realizar esta comparativa tomaremos únicamente tres tipos de dispersiones para cada máquina, intentando no saturar las gráficas. Las dispersiones elegidas son las más significativas 1, 50 y 100. Ambas arquitecturas funcionan a máximo rendimiento, esto quiere decir que el Core2Duo trabajará con sus dos núcleos mientras que el Opteron lo hará con 12.

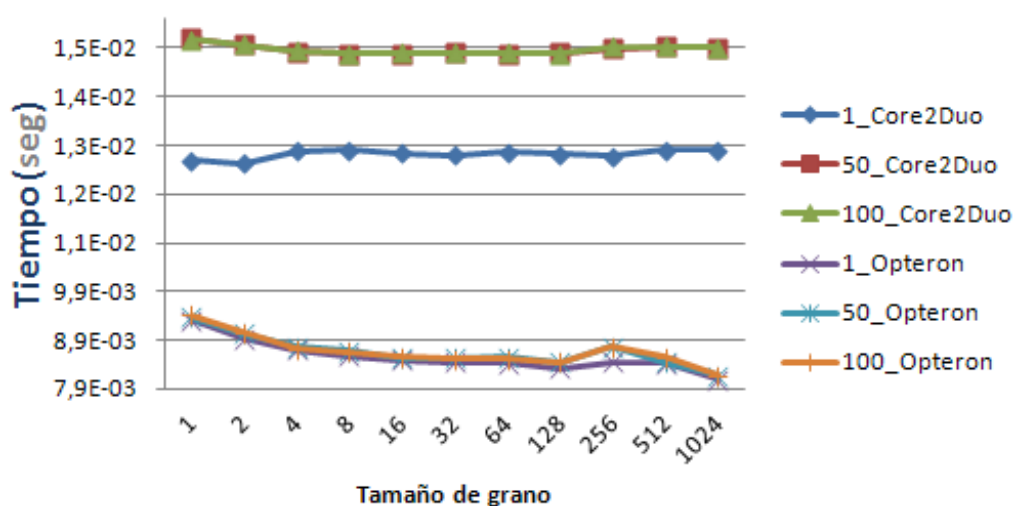
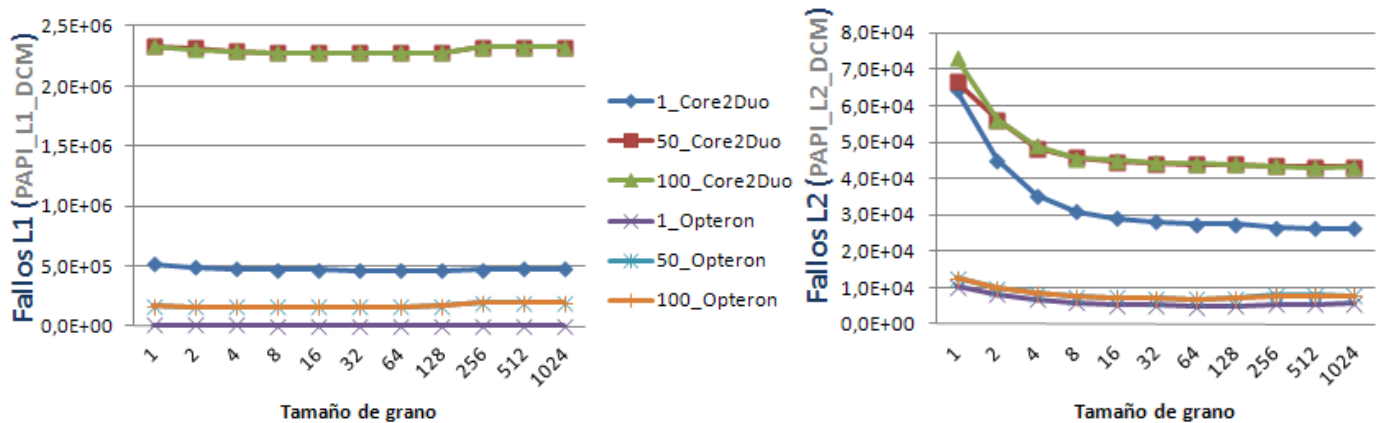


Figura 95: Tiempos para los experimentos 15 y 16 [PMV][Core2Duo y Opteron]

Los tiempos que encontramos para las distintas configuraciones muestran que el Opteron consigue resolver cómodamente en un tiempo menor al del Core2Duo. Las diferencias son tan

grandes que en la Figura 95 parece que el tamaño de grano no afecta prácticamente a los datos finales.



Figuras 96 y 97: Fallos en L1 y L2 para los experimentos 15 y 16 [PMV][Core2Duo yOpteron]

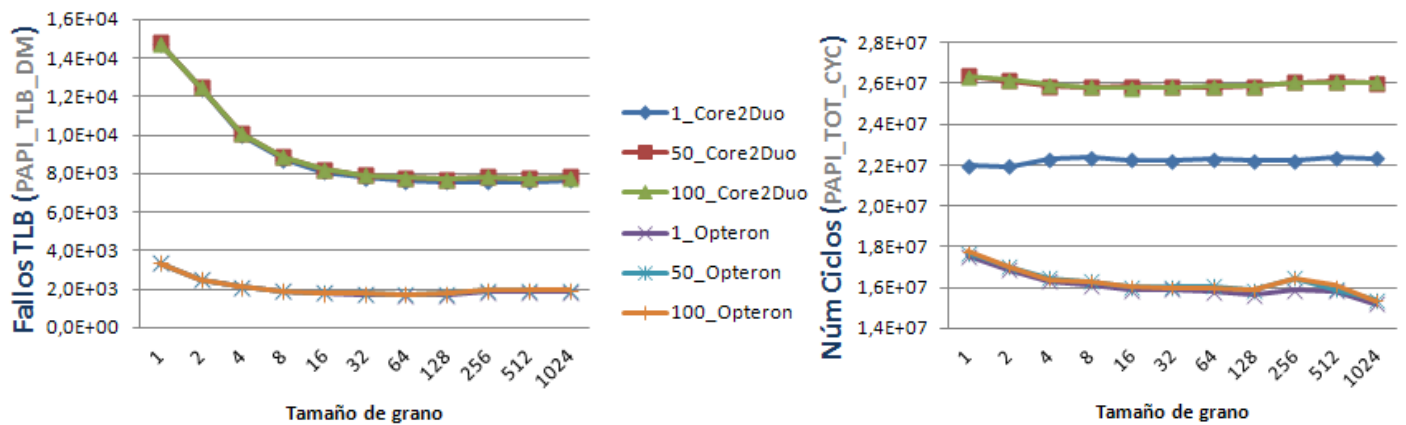
En las gráficas de fallos también se aprecia gran diferencia entre una y otra arquitectura, debemos recordar que para obtener una medida justa, tendríamos que multiplicar por 6 todos los valores que obtengamos de los contadores de PAPI, ya que en el Core2Duo mostramos la mitad de los fallos y en el Opteron solamente un doceavo. Si lo vemos de este modo los datos se asemejan bastante. Vamos a ver numéricamente si estas aproximaciones son verdaderamente reales, nos centramos en las matrices de dispersión 50:

Tamaño de grano	Fallos L1			Fallos L2		
	Core2Duo	Opteron X 6	Diferencia	Core2Duo	Opteron X 6	Diferencia
1	2332893	1015849	1317043,80	66494,4	76352,4	-9858,00
2	2318686	954642	1364044,20	56102,2	58908	-2805,80
4	2290289	972518,4	1317770,60	48263,8	51018	-2754,20
8	2279536	954380,4	1325155,20	45822,4	45424,8	397,60
16	2277241	973640,4	1303600,80	44588	43420,8	1167,20
32	2278494	969097,2	1309397,20	44143,2	42562,8	1580,40
64	2277260	969669,6	1307590,20	43941,2	41150,4	2790,80
128	2276250	1026108	1250141,60	43874,8	43080	794,80
256	2327904	1164595	1163308,80	43339,2	47295,6	-3956,40
512	2326977	1169100	1157876,80	43099,2	47464,8	-4365,60
1024	2327215	1168668	1158547,00	43212,2	46935,6	-3723,40

Tabla 9: Diferencias de fallos L1 y L2, entre Core2Duo y Opteron.

Para que nos hagamos una idea, comparamos los datos del Opteron, multiplicados por 6, con los datos del Core2Duo. Para hallar la diferencia restamos los fallos del Core2Duo menos los del Opteron. En verde tenemos los resultados en los que mejora el Opteron al Core2Duo y en rojo el caso contrario. Según estos resultados funciona mejor a nivel interno en L1 el Opteron y en L2 el Core2Duo. La comparación es claramente aproximativa y no es real del todo, ya que como hemos dicho el Opteron tiene seis veces más núcleos que el microprocesador de Intel, pero no tiene seis veces más capacidad de memoria. Es más en L2 sólo supera en 3 veces la capacidad y además no debemos olvidar el inconveniente de que es una memoria privada.





Figuras 98 y 99: Fallos en TLB y número de ciclos para los experimentos 15 y 16 [PMV][Core2Duo y Opteron]

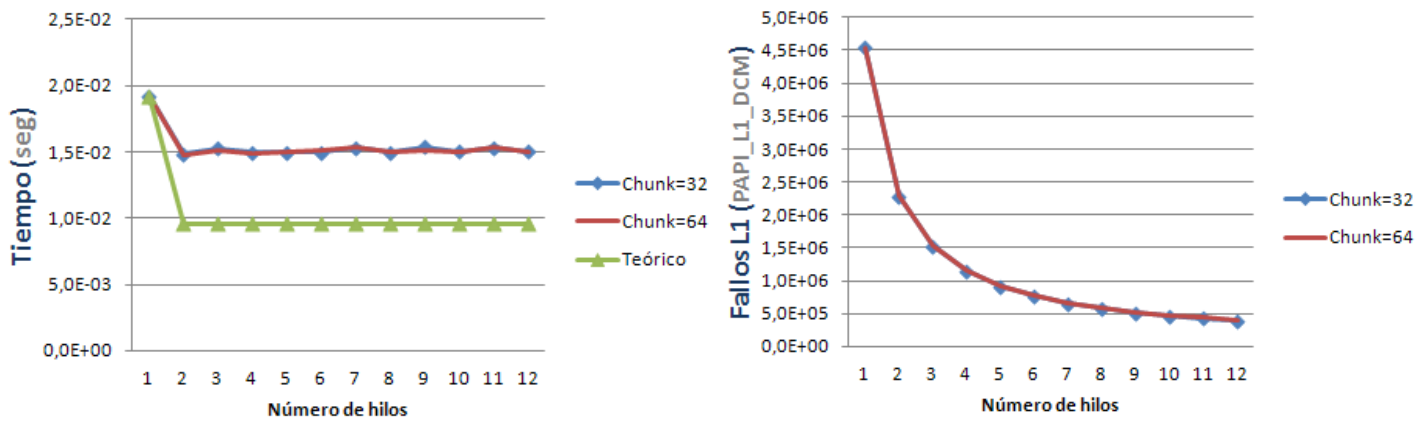
Los fallos en TLB, Figura 98, corroboran las diferencias de tiempos entre ambas arquitecturas, aunque en este caso no aparece la mejora de dispersión 1 en el Core2Duo. En cuanto al número de ciclos (Figura 99), si multiplicásemos por 6 los datos del Opteron serían mayores que los del Core2Duo, aunque como hemos comentado, muchos de los ciclos que realiza el Opteron serían en paralelo por lo que la ganancia es mayor.

#### 4.6.2 Número de hilos

Otra de las pruebas que vamos a realizar es la de ejecutar el producto matriz-dispersa vector pero variando el número de hilos. De esta forma no sólo vamos a poder comparar el funcionamiento entre el Core2Duo y el Opteron, sino que además lo vamos a poder comparar desde el punto de vista teórico. Es decir, si nuestro Core2Duo ofrece ciertos resultados con un hilo, teóricamente funcionando con dos, debería reducir los tiempos a la mitad. Es cierto, que este estudio es mucho más interesante desde el punto de vista del Opteron, ya que ofrece un mayor número de núcleos y los resultados serán más heterogéneos que el Core2Duo, al probar distintos hilos.

##### 4.6.2.1 Intel Core2Duo

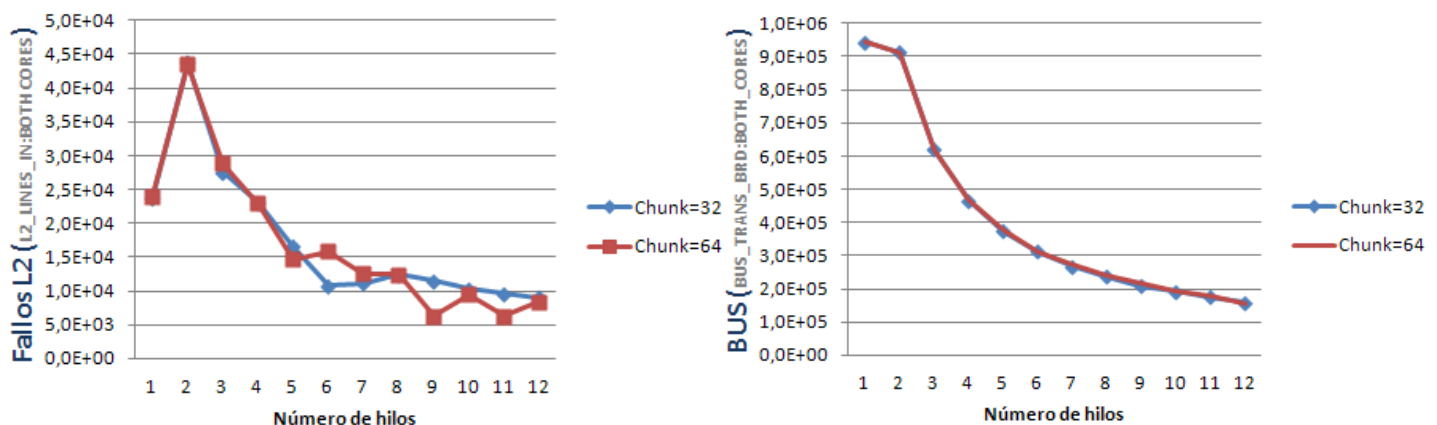
Para este experimento hemos seleccionado dos tipos de tamaños de grano similares 32 y 64 para analizar el funcionamiento con la asignación de diferentes hilos. En un principio puede parecer que los resultados que pueda ofrecer para dos hilos, serán los mismos que para 3, 4, etc. Pero fijándonos en los resultados vemos que hay ciertas diferencias. El número de repeticiones fijado para esta prueba es de 10.



Figuras 100 y 101: Tiempos y fallos en L1 para el experimento 17 [PMV][Core2Duo]

Los tiempos no cumplen las expectativas, aunque sí mejora de pasar de un hilo a dos, Figura 100, no llega a alcanzar la mitad del tiempo alcanzado con un simple hilo. En parte, tiene cierto sentido ya que al trabajar con un único núcleo, este está haciendo uso completo de la caché de nivel 2, mientras que en el resto de los casos deberá compartirla con el otro núcleo. Es cierto que se dobla el número de núcleos (con sus respectivas L1 de datos), pero no se doblan las cachés de nivel 2. Como dato curioso el sistema funciona generalmente mejor con un número de procesos pares que impares, aunque la diferencia es mínima, esto en parte puede que se deba a que tenemos un número par de núcleos. Cuando estamos ejecutando pruebas con la misma carga llegará un momento en el que uno de los procesos se estará ejecutando en un núcleo mientras el otro estará ocioso.

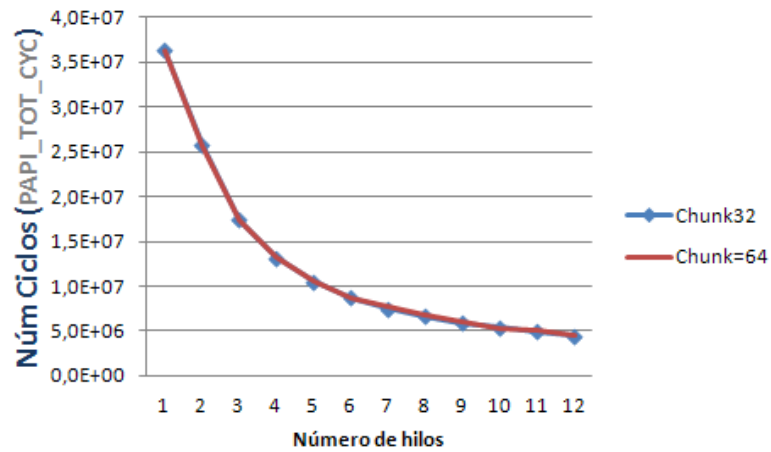
Recalcamos de nuevo que los datos que tenemos de fallos en L1, L2 o para las transacciones de bus o número de ciclos, al dividir toda la carga en N hilos, sólo estamos recogiendo los datos de ese hilo en concreto, por lo que los datos se ven divididos entre el número de hilos con el que ejecutamos la aplicación.



Figuras 102 y 103: Fallos en L2 y transferencias de bus para el experimento 17 [PMV][Core2Duo]

En este caso para los fallos en L2, Figura 102, vemos un mejor comportamiento para el tamaño de grano 64, que para 32 (las diferencias se suelen presentar en números de hilos impares). Al igual que en las pruebas anteriores para un tamaño de grano 2 obtenemos muy malos resultados.





Figuras 104: Número de ciclos para el experimento 17 [PMV][Core2Duo]

Es evidente que el número transferencias y ciclos debe ser constante para todas las ejecuciones, sin embargo, PAPI sólo nos muestra los datos de un contador por lo que las medidas no son un gran aporte para conocer si los cambios de contexto afectan realmente o no a la ejecución general del sistema. Sólo podremos en este caso tomar como referencia el tiempo, que es la única estadística tomada fuera de PAPI.

#### 4.6.2.2 AMD Opteron

Ejecutamos la misma prueba del apartado anterior, pero en el Opteron. Los resultados son los siguientes:

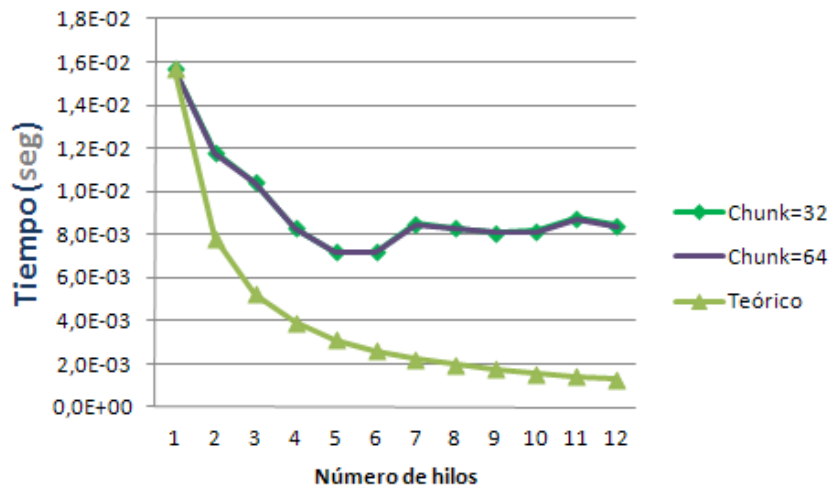
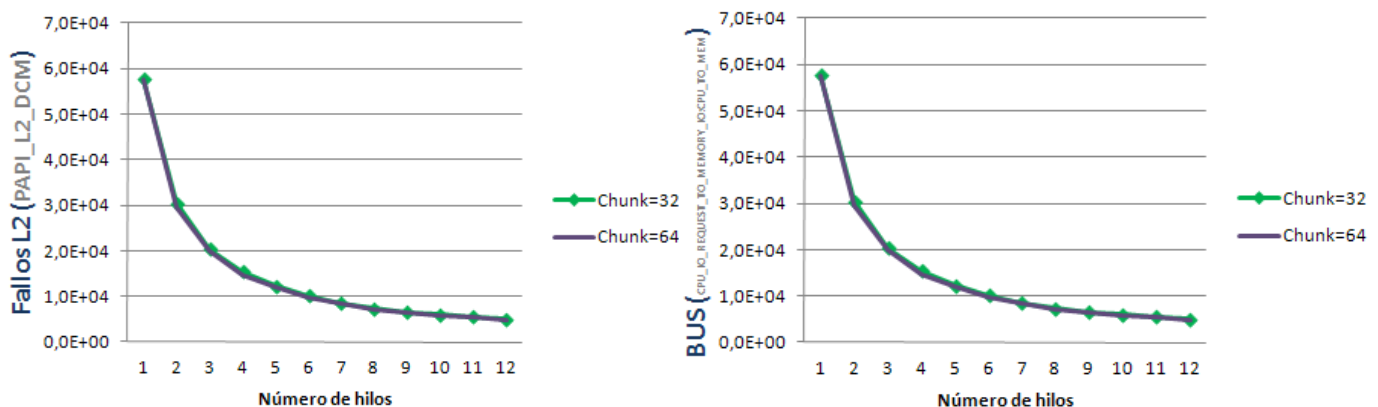


Figura 105: Tiempos para el experimento 18 [PMV][Opteron]

Calculamos de forma teórica el supuesto comportamiento que tendríamos que tener, para conseguir esta gráfica teórica (Figura 105) simplemente vamos dividiendo el valor inicial obtenido entre el número de hilos con el que se ejecuta. Sorprenden otra vez los datos, ya que la mayor ganancia la obtenemos con un número de hilos igual a 5 y 6. A partir de este punto según la prueba, no tenemos ninguna ganancia ejecutando con más hilos. Parece difícil de creer, la única posible idea que pueda hacer que con 5 o 6 hilos funcione mejor es que se esté produciendo un cuello de botella en el bus. Al solicitar tanta información los 12 núcleos, puede ser que éste se sature y no sea capaz de proveer correctamente tantas peticiones.



Figuras 106 y 107: Fallos en L2 y transferencias de bus para el experimento 18 [PMV][Opteron]

El resto de gráficas son similares. El bus a pesar de que pueda estar saturado o no (Figura 107), nos muestra el mismo efecto de bajada. Las transferencias de bus se van a mantener siempre constantes cuando los fallos sean los mismos. Sin embargo, sería interesante tomar alguna estadística de la saturación en el tiempo de esta variable, de esta forma podríamos ver que con 12 hilos el número de peticiones aumenta. Para intentar alcanzar este dato, deberíamos implementar dentro de la paralelización una toma de tiempos para saber si al principio de la prueba se producen más peticiones.

### 4.6.2.3 Comparativa

En la comparativa volveremos a ver la diferencia entre Core2Duo y Opteron. Las gráficas de ambos experimentos las comparamos a continuación para un tamaño de grano de 64.

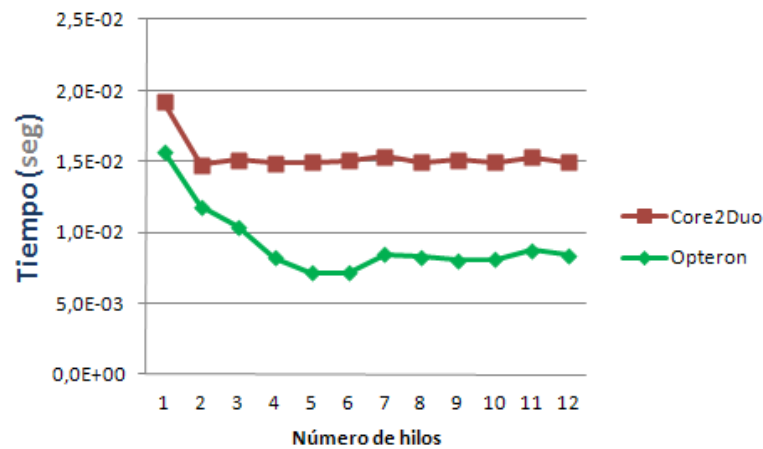


Figura 108: Tiempos para experimentos 17 y 18 [PMV][Core2Duo y Opteron]

Los tiempos que vemos en la Figura 108 son mejores para el Opteron en todas las configuraciones posibles llegando a reducir a la mitad el mejor tiempo conseguido con el Opteron. En los demás aspectos el Opteron también es superior al Core2Duo. Aunque si multiplicamos por 6 los datos obtenidos en el Opteron puede ser que en algunos casos el comportamiento del Core2Duo pueda ser más eficiente.

### 4.6.3 Tipos de matrices

Después de realizar todo tipo de comparaciones entre ambas arquitecturas vamos a continuar realizando estudios, aunque esta vez trabajaremos solamente en el Opteron. El siguiente paso que vamos a tratar será dentro de las pruebas de producto entre matriz dispersa y vector. Pero en este caso jugaremos con las matrices dispersas de la colección de la Universidad de Florida.

Dentro de la gran colección de matrices que podemos encontrar, éstas se podrían clasificar en tres grandes grupos:

- Diagonales: En las que sus elementos se encuentran principalmente en la diagonal.
- Clusterizadas: En las que los elementos se encuentran agrupadas en zonas concretas.
- Aleatorias: Donde los elementos están repartidos uniformemente por toda la matriz.

Por ello, decidimos seleccionar dos matrices de cada grupo y además como requisito añadimos que las matrices superasen el millón de elementos, cuando nos referimos a elementos queremos decir el número de valores dentro de la matriz que no valen cero. A continuación mostramos una tabla con las matrices seleccionadas, daremos un nombre corto a la matriz para que pueda ser identificado de forma más sencilla en las gráficas. En la columna de la derecha vemos la representación de las gráficas en 3D mediante algoritmos matemáticos.

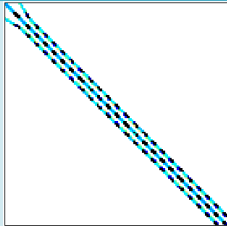
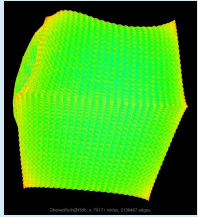
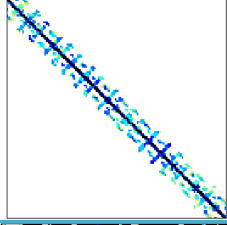
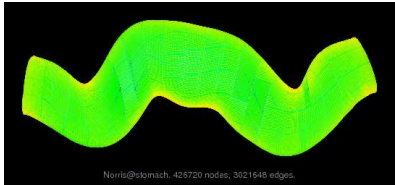
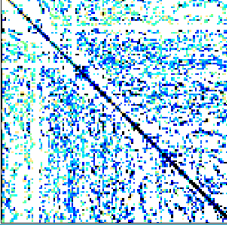
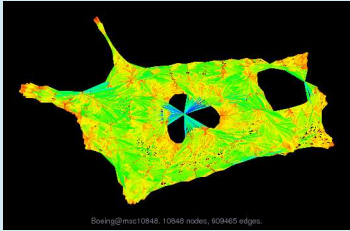
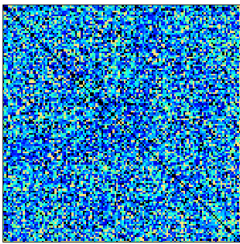
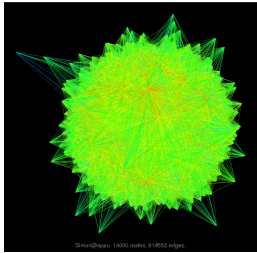
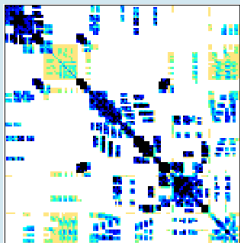
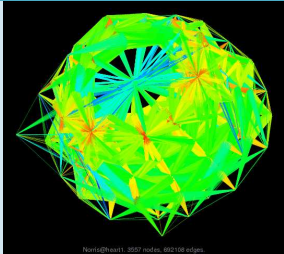
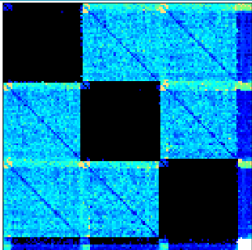
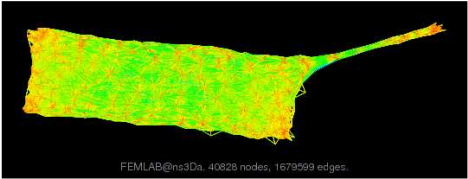
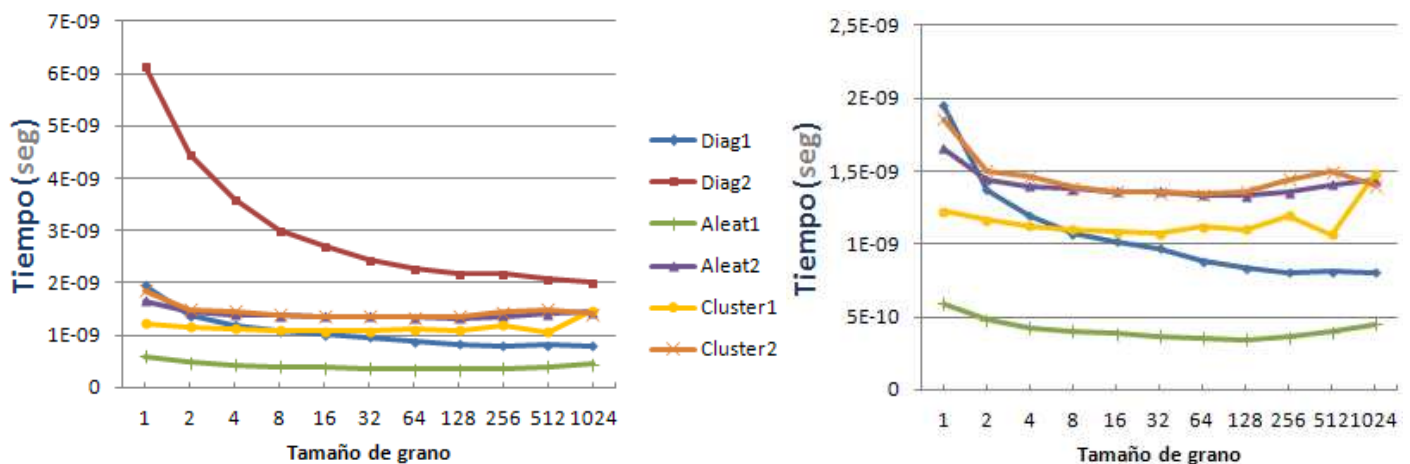
Nombre [ref]	Tipo	No ceros	Gráfico	Gráfico generado en 3D
	Nombre corto	Nº Filas		
Oberwolfach/t3dh_a [53]	Diagonal	4,352,105		
	Diag1	79,171		
Norris/stomach [54]	Diagonal	3,021,648		
	Diag2	213,360		
Boeing/msc10848 [55]	Aleatoria	1,229,776		
	Aleat1	10,848		
Simon/appu [56]	Aleatoria	1,853,104		
	Aleat2	14,000		
Norris/heart1 [57]	Clúster	1,385,317		
	Cluster1	3,557		
FEMLAB/ns3Da [58]	Clúster	1,679,599		
	Cluster2	20,414		

Tabla 10: Matrices a utilizar para la comparativa.

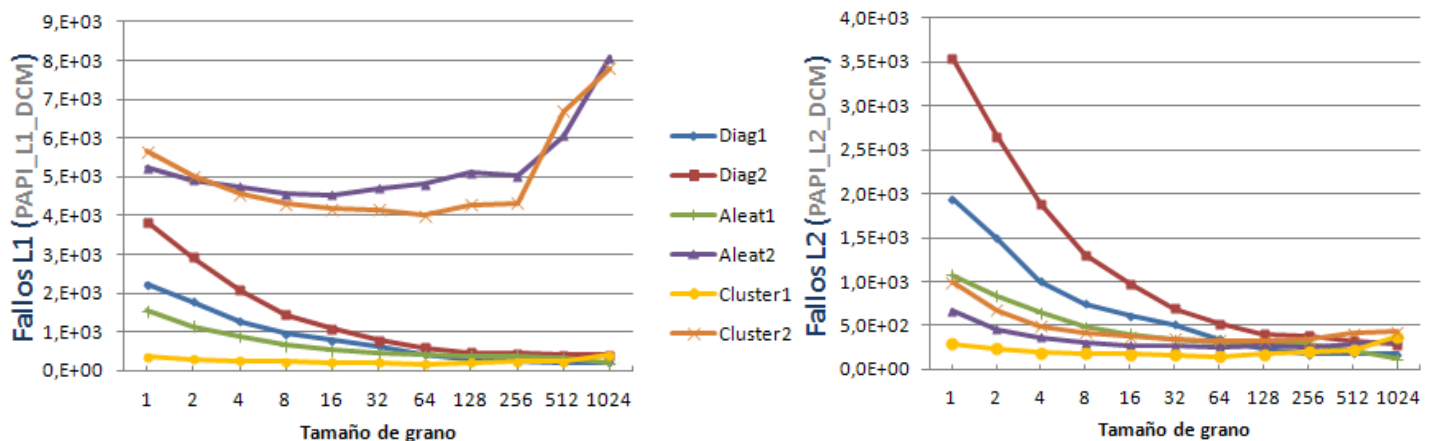
Todas las matrices utilizadas tienen el mismo número de filas que de columnas. Además todas ellas almacenan valores que pertenecen al conjunto de los números reales. El resultado final se espera que sea similar al obtenido con las matrices de dispersión en el Apartado 4.5.1.

En las siguientes gráficas veremos los resultados que producen la multiplicación de estas matrices por el mismo vector que hemos utilizado anteriormente, progresión aritmética de incremento +1, comenzando en la unidad (1, 2, 3, 4, 5, 6...).



Figuras 109 y 110: Tiempos para el experimento 19 con/sin Diag2 [PMV][Opteron]

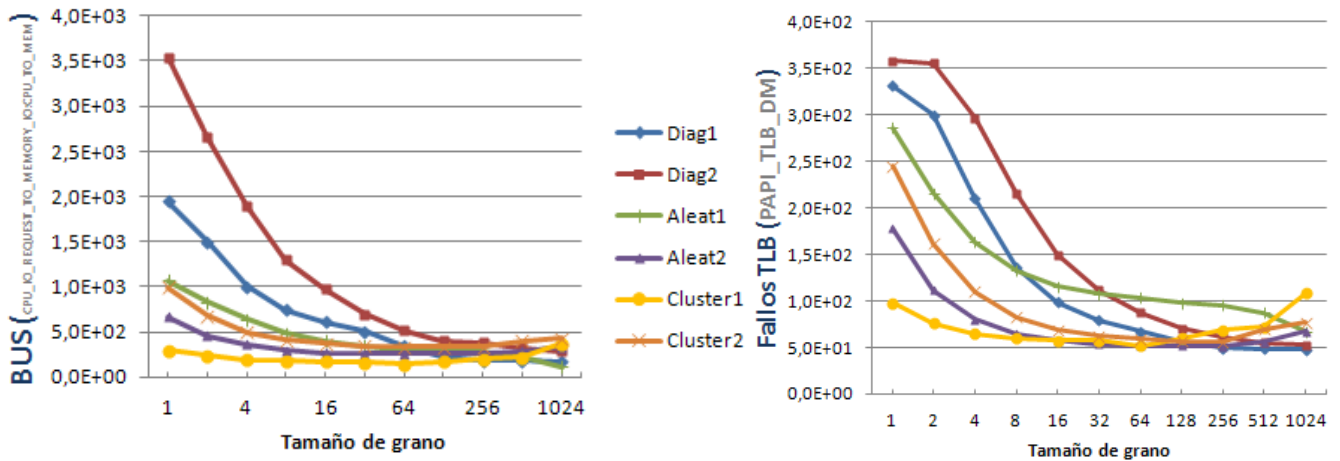
Mostramos las gráficas con Diag2 (Figura 109) y sin ella (Figura 110), puesto que sus resultados distan bastante. A pesar de dividir los resultados por el número de elementos no cero de cada matriz (normalización), la matriz de mayor tamaño es la que más tiempo tarda en ejecutar. Esto nos dice que la relación entre tiempo y número de elementos no es directamente proporcional y, que a un mayor número de elementos la función de tiempo crece de forma exponencial. Otra premisa que nos puede llevar a esta conclusión es que la matriz con menos elementos, a pesar de ser aleatoria, es la que está mostrando un mejor comportamiento.



Figuras 111 y 112: Fallos en L1 y transferencias de bus para el experimento 19 [PMV][Opteron]

En la gráfica de fallos en L1 (Figura 111) será la única en la que las matrices diagonales se vean favorecidas con respecto a alguna de las otras. Las matrices Aleat2 y Cluster2, que como vimos en las gráficas, son bastante dispersas son las que muestran una tasa de fallos superior, da igual qué tamaño de grano utilicemos que estas matrices no consiguen contener el vector de multiplicación en la caché de nivel 1 y, por consiguiente, obtenemos continuos fallos de memoria caché. En la gráfica de fallos en L2 (Figura 112) vemos un comportamiento muy usual, en él las dos matrices de mayor tamaño no consiguen entrar en L2. Al tener un

tamaño de grano fino aparece un efecto claro de *false sharing* en el vector de escritura. Si estudiamos más detenidamente lo que ocurre con este vector, tenemos lo siguiente: es un vector de tipo real, el tamaño de un valor real es de 8 Bytes en esta arquitectura, por lo que si el tamaño de bloque es de 64 Bytes, sabemos que en un bloque van a caber 8 valores. Por tanto para un tamaño de grano 1 serán 8 los núcleos que compartan un mismo bloque. Para un tamaño de grano 8 estos vectores deben de dejar de compartir datos, suponiendo que el vector comience en un bloque nuevo, opción que está activada en la compilación con el valor -O3.



Figuras 113 y 114: Peticiones el bus de memoria y fallos TLB para el experimento 19 [PMV][Opteron]

La gráfica de peticiones al bus de memoria (Figura 113) no muestra nada nuevo, ya que las matrices tampoco caben en la memoria L3, por tanto el número de peticiones está directamente con el número de fallos en L2.

Los fallos de TLB (Figura 114) son menores a menor tamaño de grano y este aumenta conforme el tamaño de grano es más grueso. Como hemos dicho en otros experimentos este efecto de aumento y bajada se debe a que en los tamaños de grano pequeños, si lo vemos desde el punto de vista de un núcleo, éste va a tener una página llena de datos que él no utilizará pero otros núcleos sí. Según aumente el tamaño de grano, cada vez las páginas de memoria virtual contendrán más información concerniente a ese núcleo y por tanto irá reduciendo de forma progresiva la tasa de fallos en TLB.

Vamos a ver un ejemplo práctico con la Figura 115. Primero calculamos el tamaño medio de elementos por fila que tienen estas matrices dividiendo número de no-ceros entre número de filas. Esto hace un total de: 54.9, 14.1, 133.3, 132.3, 389.4 y 82.2 respectivamente. Si suponemos una matriz con 100 elementos no-cero por fila tenemos lo siguiente. En gris se muestran las posiciones accedida por núcleo. En separación en verde se muestra las divisiones por tamaño de página.

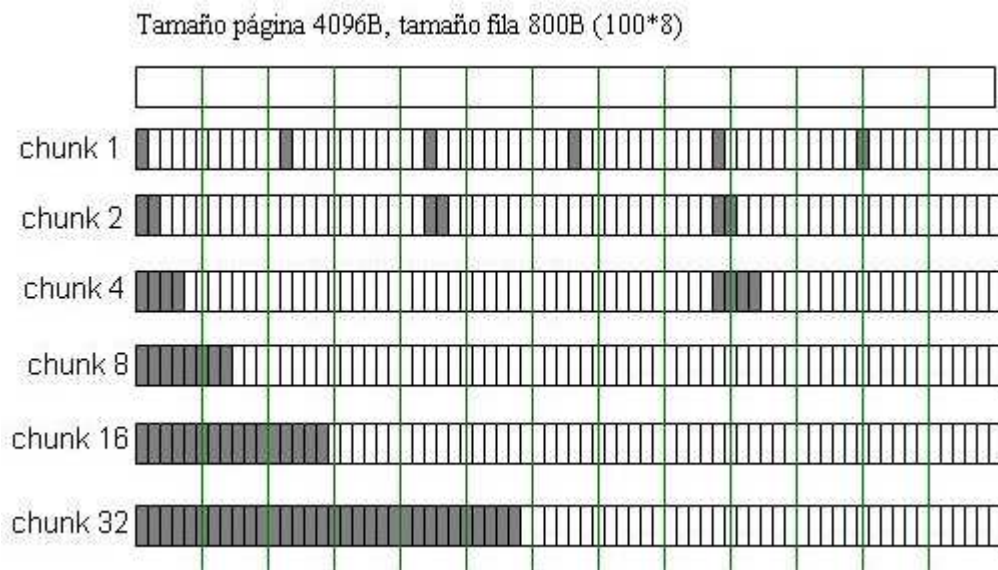


Figura 115: Explicación a los fallos en TLB por tamaño de grano fino

Luego en tamaño de grano 1 y 2, se ve cómo hay un gran desperdicio de páginas que contienen información que no utilizarán y que tendrá que cargar una de cada dos páginas para obtener la información. Para tamaños de grano mayores el porcentaje de datos que cargamos no útiles es menor.

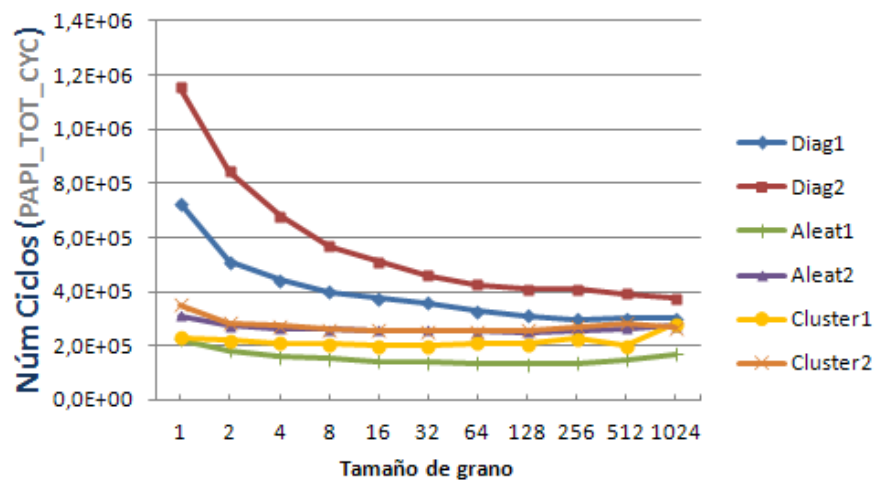


Figura 116: Número de ciclos para el experimento 19 [PMV][Opteron]

Para finalizar la gráfica de número de ciclos (Figura 116) muestra menor número de diferencia comparando con los tiempos. Aunque seguimos viendo como para matrices mayores el número de ciclos es superior aún dividiendo este valor entre el número de elementos no cero.



#### 4.6.4 Tipos de planificación

Otra de las pruebas interesantes que nos concede OpenMP es probar con distintos tipos de planificación o distintas formas de repartir la carga de computación. Estos tipos son:

- *estática o static*: en este caso, todos los hilos tienen asignada su carga de trabajo antes de que comiencen las iteraciones. La carga de iteraciones se divide de forma igual por defecto. Sin embargo, si se especifica el tamaño de grano, esta división se realizará a partir de éste y no entre el número de hilos.
- *dinámica o dynamic*: aquí se realizará un reparto inicial de las tareas y, una vez que un hilo finalice su tarea, se le asigna una nueva de las que quedan por completar. El parámetro *chunk* en este caso define el número de iteraciones contiguas de las que se encargará el hilo cada vez que haya un reparto.
- *guiada o guided*: un gran número de iteraciones contiguas se destinan a cada hilo de forma dinámica (como en el modo *dynamic*). El tamaño de grano decrece con cada sucesiva asignación, finalizando en un tamaño mínimo, que es el dado por el parámetro *chunk*.

Las pruebas que hemos realizado hasta ahora han sido siempre con una planificación estática. Compararemos las ejecuciones realizadas con nuevas utilizando los otros dos tipos de planificación restantes. Utilizaremos la mayor matriz que hemos utilizado hasta ahora Diag1 (Oberwolfach/t3dh\_a) y mantendremos la división de los resultados del contador entre.

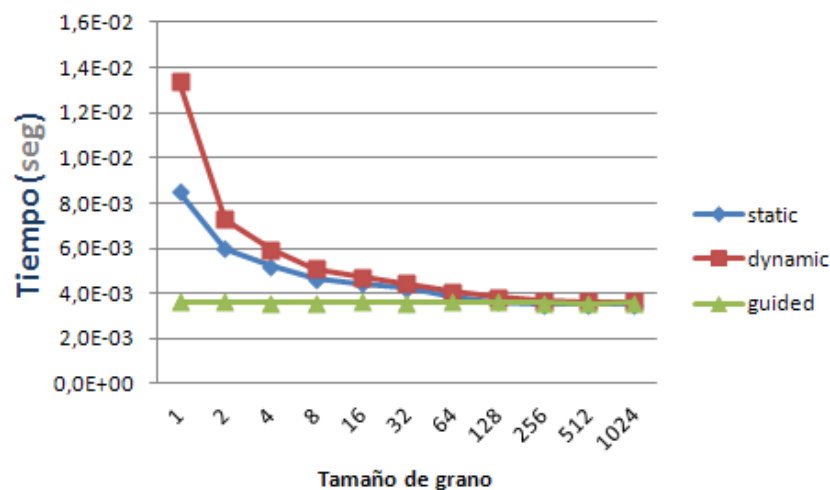


Figura 117: Tiempo en segundos para el experimento 20 [PMV][Opteron]

Los resultados muestran (Figura 117) que a partir de cierto tamaño de grano las planificaciones se ejecutan de forma similar. Si bien es cierto que la opción *guided* ofrece el mismo resultado durante todos los experimentos independientemente del tamaño de grano asignado. Según la información que tenemos de esta opción se le da un tamaño de grano inicial que decrece hasta tomar el valor indicado por el *chunk*. Sin embargo, no concreta cual es el valor inicial dado. Después de introducirnos un poco en el código y ver qué iteraciones se asignan a cada hilo, no podemos sacar una conclusión clara, ya que en una de las ejecuciones con un tamaño de grano 32 tenemos:



Hilo	0	1	2	3	4	5
Primera iteración asignada	0	312238	617036	1345254	872247	1180575
Hilo	6	7	8	9	10	11
Primera iteración asignada	1266604	472252	1086693	750450	983965	139338

Tabla 11: Iteraciones asignadas por hilo con planificación *guided*, tamaño de grano 32

Hemos intentado sacar algún tipo de relación a estos números pero no existe ninguna directa. Por lo que no estamos muy seguros del funcionamiento de esta última opción. Es probable que el tamaño de grano dado inicialmente sea muy grande y nunca llegue a descender a menos de 8 o 16. De esta forma se explicaría este comportamiento ya que para el tamaño de grano 128 y los consecutivos las gráficas se igualan.

Si observamos el mismo ejemplo con planificación *dynamic* o *static* tenemos los mismos resultados. Todos los números son múltiplos de 16 (deberían ser múltiplos de 32 también), y el orden de asignación es más coherente, de menor a mayor.

Hilo	0	1	2	3	4	5
Primera iteración asignada	0	288	768	1056	1440	1920
Hilo	6	7	8	9	10	11
Primera iteración asignada	2480	2880	3344	3888	4512	5088

Tabla 12: Iteraciones asignadas por hilo con planificación *static* ó *dynamic*, tamaño de grano 32

### 4.7 Huge pages

En este caso utilizamos el Core2Duo para realizar una comparación entre la utilización o no de la utilidad que nos proporciona la funcionalidad de las huge pages. Para realizar este experimento tomamos como ejemplo el número 9, en el que ejecutamos *false sharing* sobre una matriz de 3 MBytes. Reservamos 4 páginas como comentamos en el Apartado 2.4.4 y procedemos a ejecutar la misma prueba. Los resultados son los siguientes:

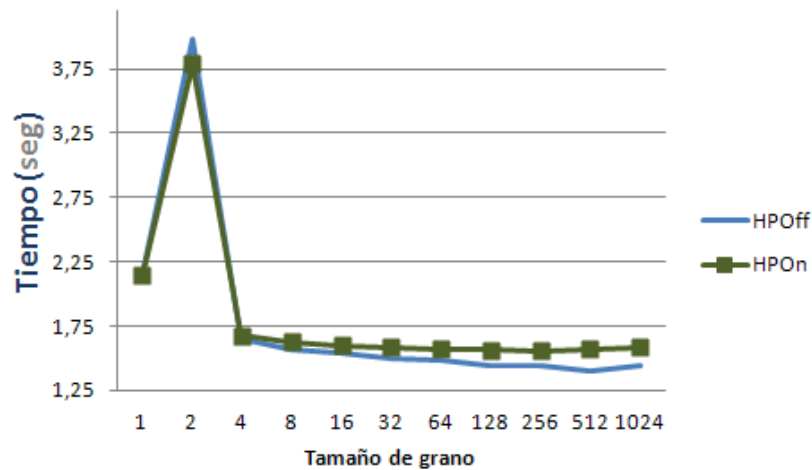
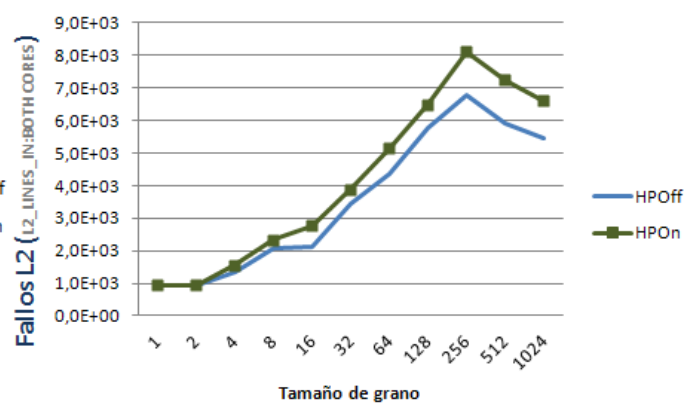
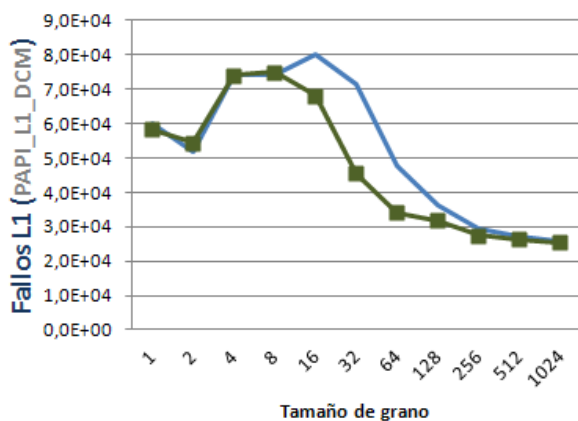


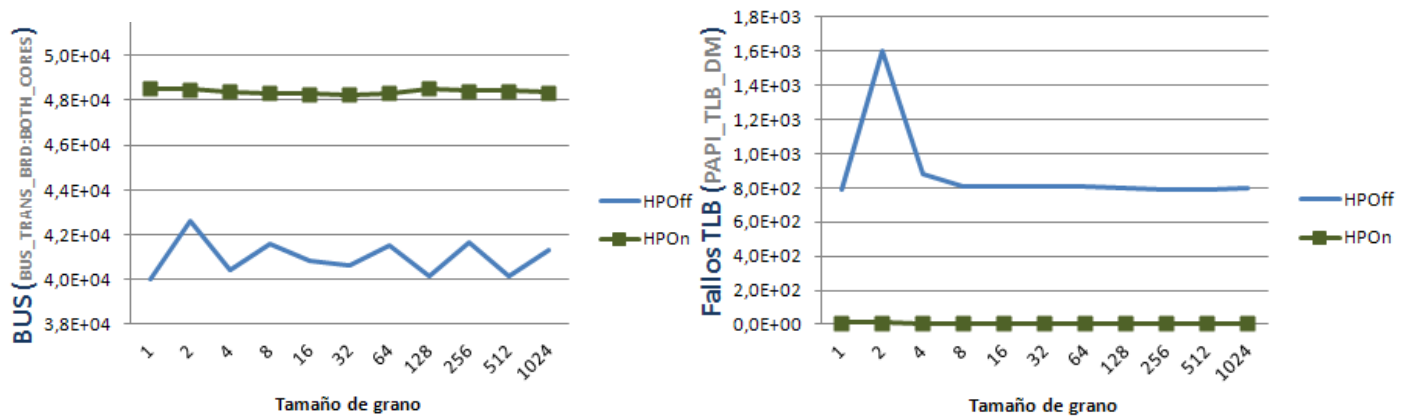
Figura 118: Tiempo en segundos para los experimentos 9 y 21 [FS][Core2Duo]

Después de la primera ejecución los resultados no son muy alentadores (Figura 118). Excepto en los dos primeros casos en el resto de ellos consigue realizar las pruebas en menor tiempo sin activar las opciones de Huge Pages. En las siguientes gráficas veremos el porqué de esto.



Figuras 119 y 120: Fallos en L1 y L2 para los experimentos 9 y 21 [FS][Core2Duo]

En la gráfica en L1 (Figura 119) sí vemos como se comporta mejor con la utilización de huge pages, pero supone un lastre para L2 (Figura 120). Tenemos que pensar que al dejar de producirse los fallos en la TLB, la dirección de memoria virtual se traduce a memoria física de forma correcta, sin embargo cuando accede a la memoria L2, la página no se va a encontrar completamente en memoria ya que algunos bloques de memoria, los que hayan sido utilizados hace más tiempo, habrán sido reemplazados por los nuevos datos.



Figuras 121 y 122: Transferencias de bus y fallos en TLB para los experimentos 9 y 21 [FS][Core2Duo]

En estas dos gráficas se ve claramente las ventajas e inconvenientes de la utilización de huge pages. Como gran inconveniente, y probablemente principal causa de que los tiempos sean mayores con huge pages, el bus de datos (Figura 121) se satura bastante más, al producirse más fallos en L2. Tenemos que tener en cuenta que si se produce un fallo TLB la página y el correspondiente bloque serán cargados, sin embargo, después de este proceso puede que, a pesar de que la página se encuentre (aciertos TLB) los datos no estén en las cachés y esto repercute directamente en los tiempos. Por otra parte, la gráfica de fallos TLB (Figura 122) no hace más que corroborar el correcto funcionamiento de huge pages, consiguiendo reducir los fallos TLB a una cifra aproximada de 7.05 fallos por ejecución. Cifra que queda muy lejos de los 879 fallos sin la utilización de huge pages.

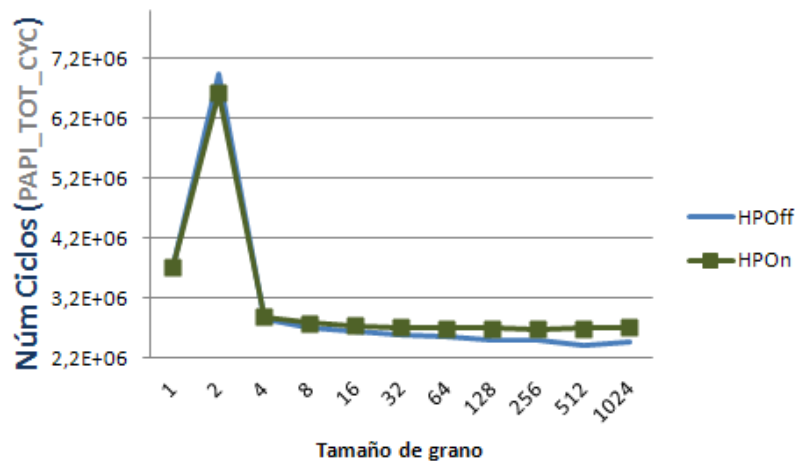


Figura 123: Número de ciclos para los experimentos 9 y 21 [FS][Core2Duo]

Para finalizar vemos cómo es necesario realizar un mayor número de ciclos para alcanzar el mismo resultado con huge pages, que sin ello (véase Figura 123). Por lo tanto, queda demostrado que la utilización de esta técnica para esta prueba y para estos tamaños de grano en concreto no es eficiente.

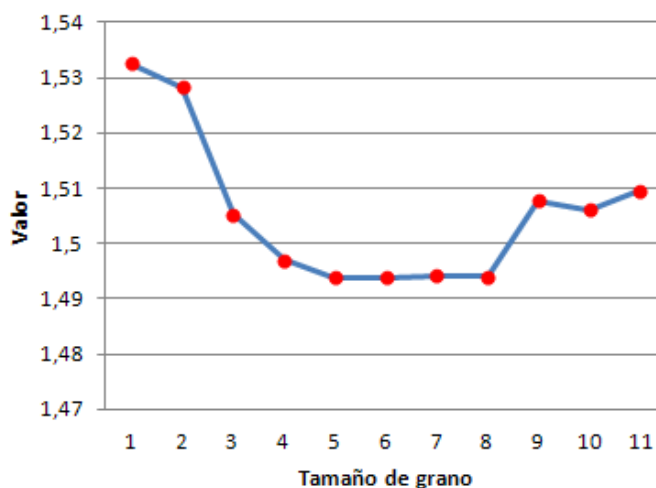
## 4.8 Integración con Técnicas de optimización

Si nos centramos de nuevo en el producto matriz vector, hasta ahora sabemos que una planificación estática y que tamaños de grano altos son mejores a la hora de reducir tiempos. No obstante, esto último no se puede corroborar al cien por cien ya que no hemos realizado pruebas para todos y cada uno de los tamaños de grano, algo que requeriría mucho tiempo de computación. Para intentar conocer cuál podría ser el posible mejor tamaño de grano vamos a utilizar dos técnicas de búsqueda que se acercarán de forma precisa al mejor tamaño de grano posible o, dicho de otra forma, al mejor mínimo global de la función.

Debido a que queremos realizar unas primeras pruebas para testear estas técnicas y variar sus diferentes valores, hasta alcanzar una configuración más eficiente, hemos creado un pequeño método por el cual simplemente indicando unos puntos de una función, nos retornará cualquier valor posible de dicha función. La función se formará mediante la unión de estos puntos de tal forma que si en el eje x el punto 5 vale 5 y el punto 7 vale 7, al pedirle al método el valor en el punto 6 nos retornará 6. Para ello utilizamos la fórmula que genera una recta a partir de dos puntos:

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} \rightarrow y = \frac{(y_2 - y_1)(x - x_1)}{x_2 - x_1} + y_1$$

Siendo (x1, y1) las coordenadas de un punto y (x2, y2) las coordenadas del otro. El valor de entrada sería "x" y la salida "y". De esta forma a partir de un conjunto de número tendremos una función formada por rectas que nos permitirá ahorrar tiempo a la hora de configurar los parámetros iniciales de estos dos métodos.



Con los puntos: 1.5325, 1.5282, 1.5053, 1.497, 1.4939, 1.4938, 1.4942, 1.4941, 1.5077, 1.5061 y 1.5095 generamos la función que aparece en la Figura 124.

Figura 124: Función creada a partir de 11 puntos

En los siguientes apartados experimentaremos con ambas arquitecturas y comentaremos los resultados obtenidos de forma resumida.

Las gráficas que tenemos inicialmente en el caso del Core2Duo poseen zonas donde existen mínimos locales, algo interesante a la hora de probar los algoritmos, sabemos que tanto las que mostramos a continuación como las del Opteron no tendrán ninguna relación con las gráficas finales ya que en estos casos estamos obviando gran parte de los resultados intermedios entre los distintos tamaños de grano.

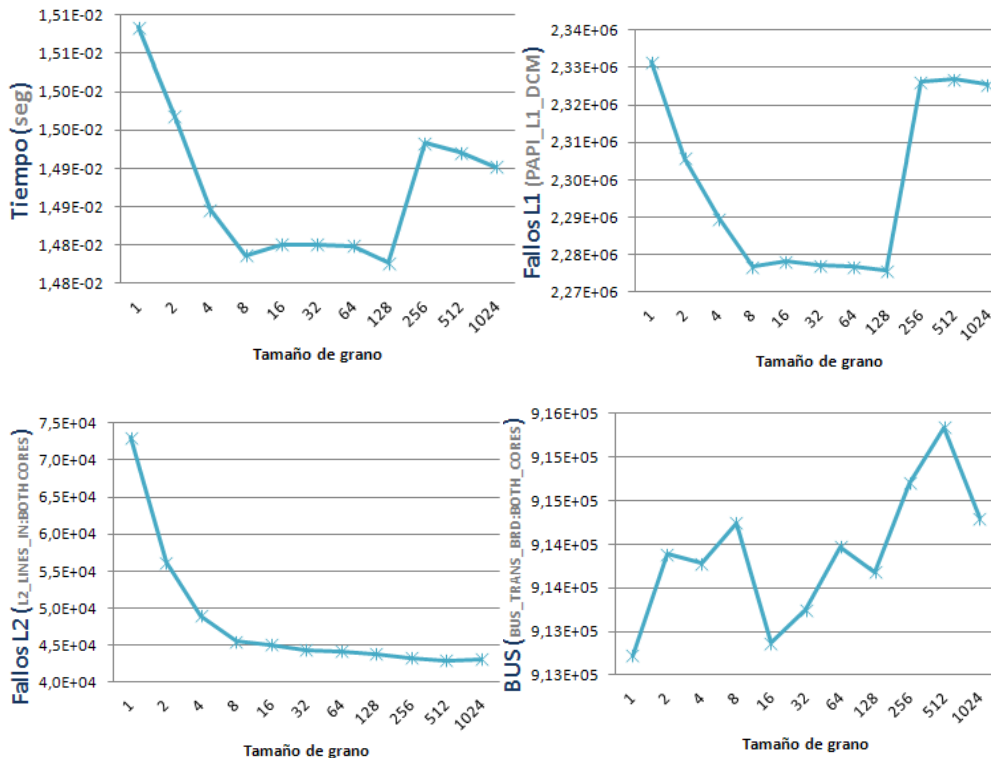


Figura 125: Gráficas seleccionadas para la experimentación [PMV][Core2Duo]

Como dato a favor de estos algoritmos debemos saber de antemano que esta gráfica no es exacta, debido a la gran complejidad de las arquitecturas actuales nunca una ejecución será igual a otra y, por tanto, puede ocurrir que dos ejecuciones con mismo tamaño de grano ofrezcan resultados distintos, por ello y para asegurarnos el menor error posible cada ejecución de benchmark supondrá 5 ejecuciones de la misma prueba y, la salida que obtendremos al final será la media de estas ejecuciones.

Las pruebas se realizarán sobre el producto matriz dispersa – vector, para una dispersión total del 100%. En este apartado no nos encargaremos de comparar dos arquitecturas, si no dos técnicas de optimización. Iremos viendo los resultados ofrecidos por ambas técnicas.

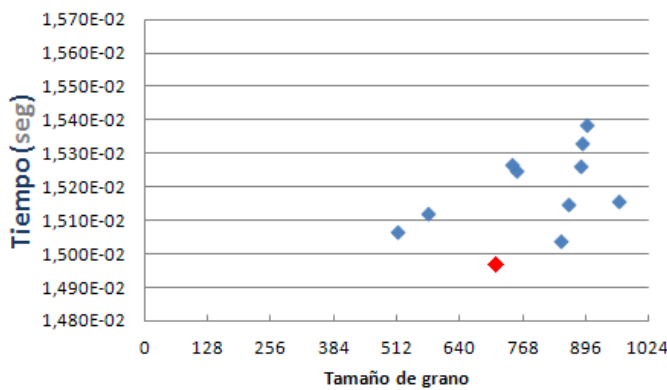
#### 4.8.1 Simulated Annealing

Para configurar los parámetros de *Simulated Annealing* utilizamos la función generada por puntos, puesto que se genera una respuesta automática a una entrada y no tenemos que ejecutar el benchmark para probar los parámetros de entrada. Modificamos la temperatura mínima, máxima, el tamaño de salto, número de intentos, etc... hasta conseguir una configuración que consiga alcanzar el mejor mínimo en el menor número de pasos. Sin embargo, esta configuración es un poco relativa ya que cada función tendrá una configuración de búsqueda óptima que es imposible de conocer a menos que realicemos todas las pruebas

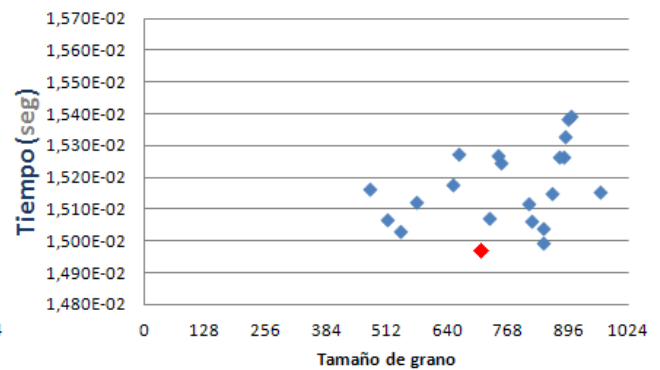
para todos los tipos de configuraciones, algo inviable debido al gran número de combinaciones posibles.

El punto de partida de *Simulated Annealing* lo fijamos en la mitad de la gráfica (512), de esta forma valorará las opciones de saltar hacia la parte izquierda o derecha de la función. En las siguientes gráficas veremos los distintos saltos con distinto número de iteraciones, cada iteración supone un total de 10 ejecuciones del benchmark, sin contar la inicial.

### 1 Iteración



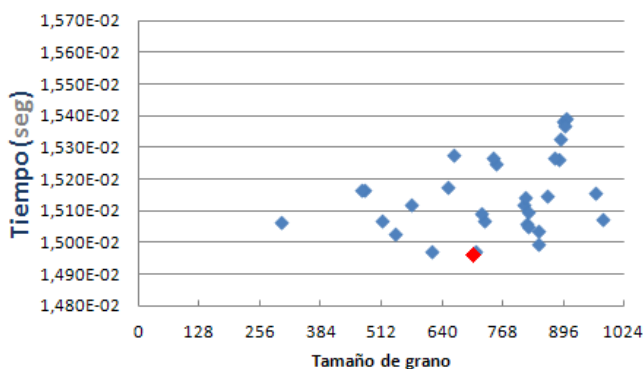
### 2 Iteraciones



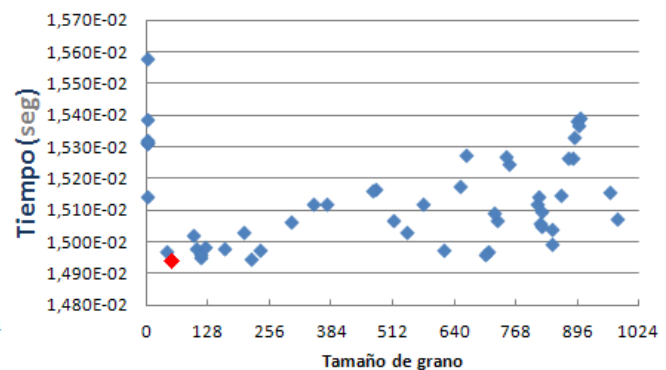
Figuras 126 y 127: Puntos testeados por *Simulated Annealing*, 1 y 2 iteraciones [PMV][Core2Duo]

Cuando sólo hemos realizado una iteración se alcanza un buen mínimo con un tamaño de grano 711 (Figura 126), el algoritmo inicialmente se centra en la parte derecha de la función.

### 3 Iteraciones

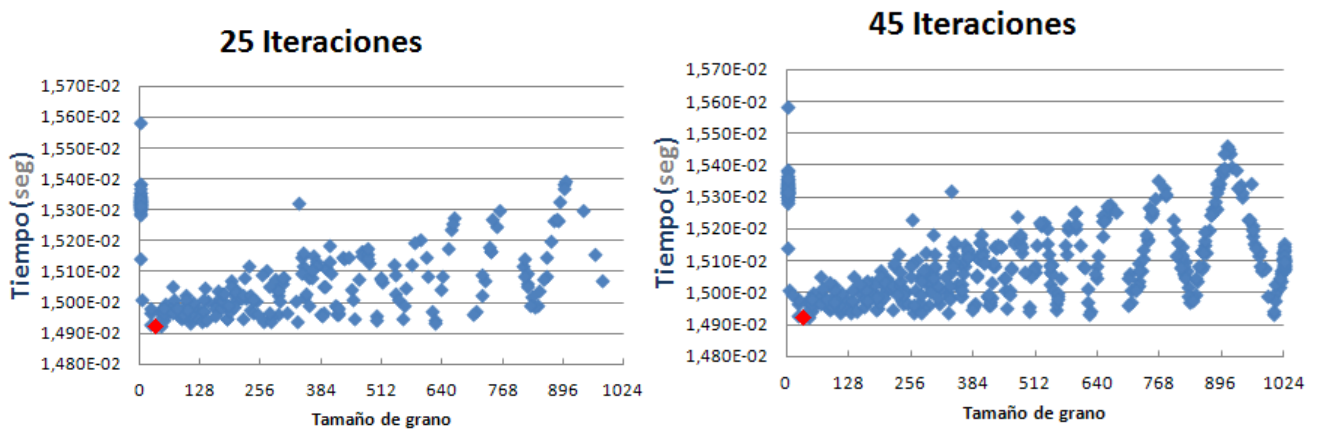


### 5 Iteraciones



Figuras 128 y 129: Puntos testeados por *Simulated Annealing*, 3 y 5 iteraciones [PMV][Core2Duo]

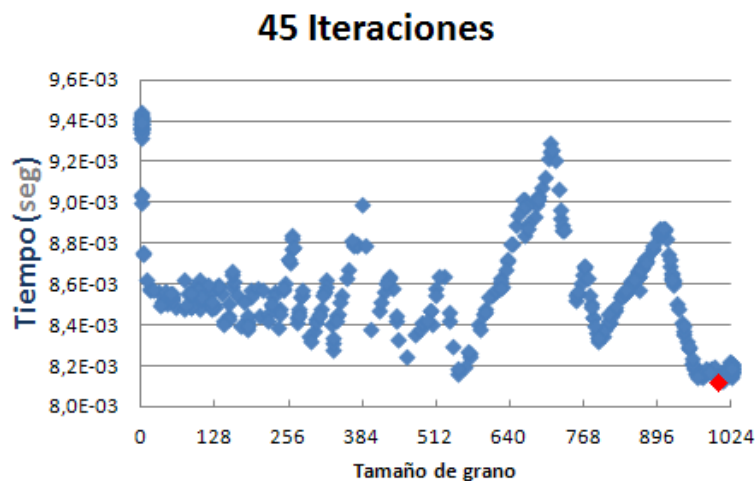
Según aumentan el número de iteraciones y se va disminuyendo la temperatura inicial, comprobamos que se alcanzan varios mínimos globales nuevos en 704, 218 y 52 (Figuras 128 y 129). Cabe destacar que la función de *Simulated Annealing* es capaz de predecir saltos negativos y menores que 1, por eso hemos tenido que limitar esta posibilidad para acotar el rango de búsqueda y porque no tiene sentido un tamaño de grano menor de 1. Por lo tanto, el futuro salto siempre tiene que ser mayor o igual a 1 y menor o igual a 1024. Esto último explica el porqué existan más de una ejecución con un mismo tamaño de grano.



Figuras 130 y 131: Puntos testeados por Simulated Annealing, 25 y 45 iteraciones [PMV][Core2Duo]

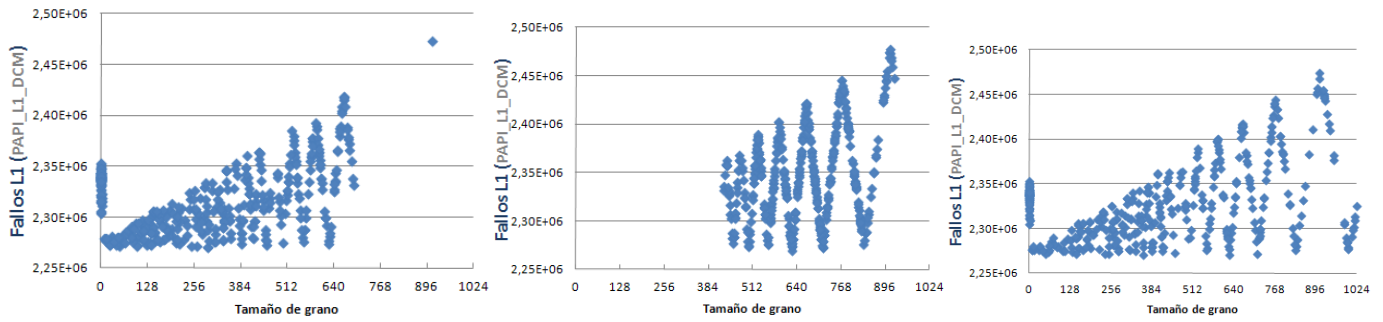
Finalmente, después de ejecutar 45 iteraciones, el mejor mínimo se encontró en la iteración 6, con un tamaño de grano 33 (Figura 131). La función está prácticamente definida después de 45 iteraciones y es que tenemos que tener en cuenta que hemos ejecutado un total de 450 pruebas más la inicial, en total *Simulated Annealing* ha repetido 122 pruebas que ya había realizado, luego hemos evaluado un 32,12 % del total.

Como curiosidad en el Opteron también aparece una gráfica similar aunque más pronunciada, en este caso encontramos el mejor mínimo global después de la iteración 17 con un tamaño de grano 1002. En total se realizan 157 repeticiones de pruebas ya ejecutadas, luego se ejecuta un 28,71% del total.

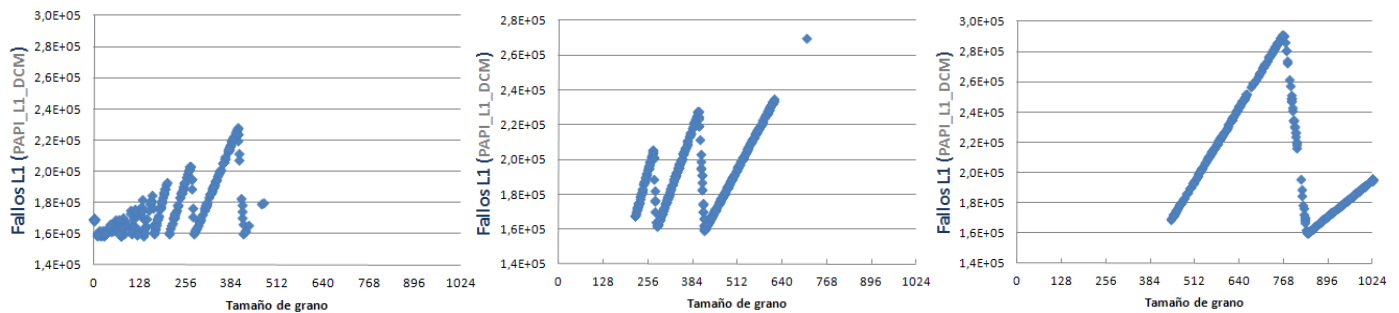


Figuras 132 y 133: Puntos testeados por Simulated Annealing, 45 Iteraciones [PMV][Opteron]

Sin embargo, en algunas gráficas nos vamos a encontrar un problema hasta ahora no visto, *Simulated Annealing* se centra en unas zonas y no explora todo el espectro de la gráfica. Las siguientes imágenes son las generadas para fallos en L1 en el AMD Opteron, después de 45 iteraciones. Para obtener estas gráficas tenemos que variar el lugar en el que comienzan a realizar las pruebas, que es donde parece que más se centra *Simulated Annealing* y además cambiar el tamaño de salto. A mayor tamaño más amplitud de exploración.



Figuras 134, 135 y 136: Gráficas generadas por Simulated Annealing con distintas configuraciones para fallos en L1 después de 45 Iteraciones [PMV][Core2Duo]



Figuras 137, 138 y 139: Gráficas generadas por Simulated Annealing con distintas configuraciones para fallos en L1 después de 45 Iteraciones [PMV][Opteron]

Si unimos estas gráficas podemos tener la visión general de lo que está ocurriendo con los fallos en L1 respectivamente para Core2Duo y Opteron:

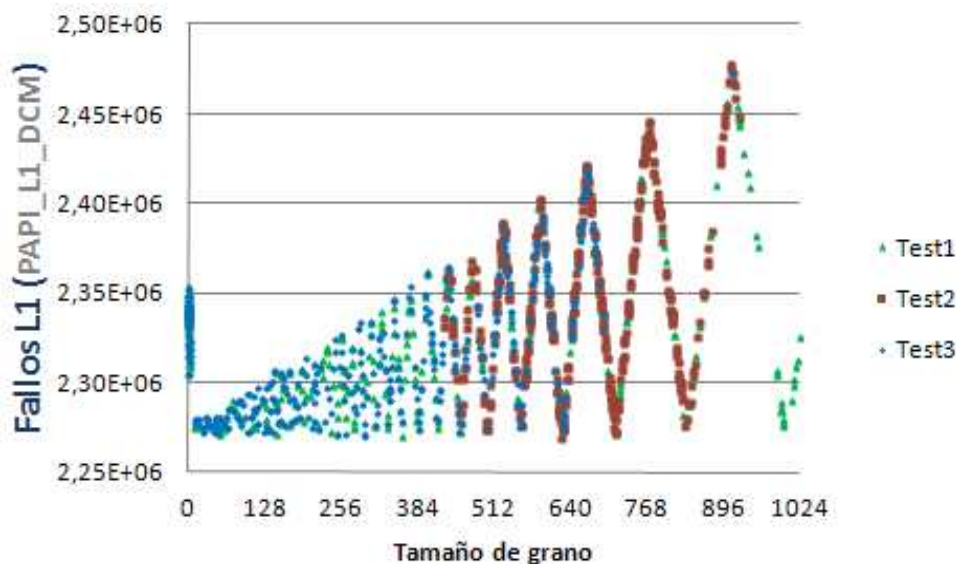


Figura 140: Muestra de los tres test en la misma gráfica [PMV][Core2Duo]



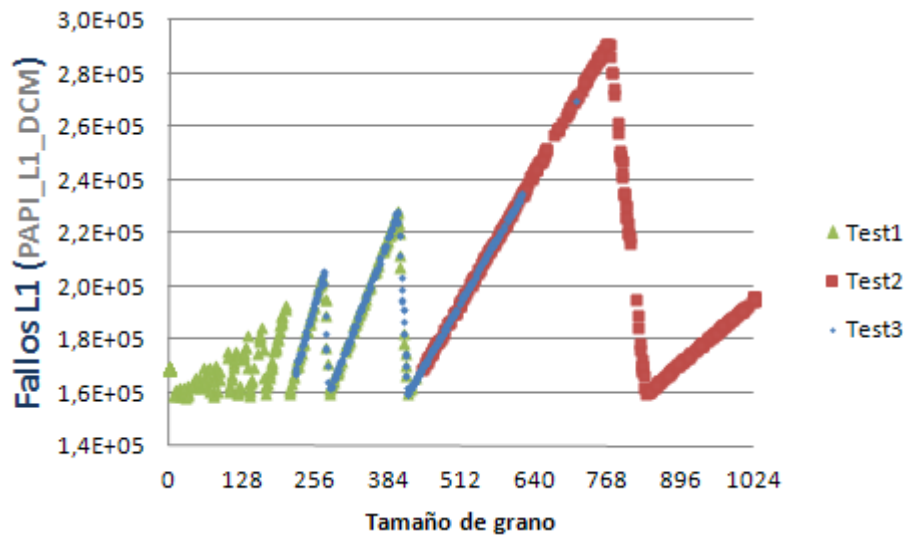


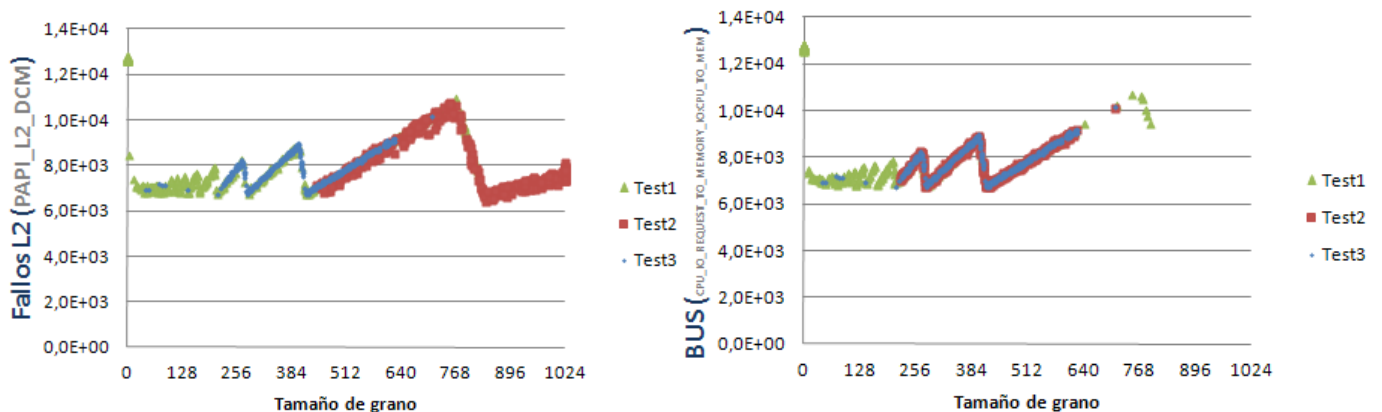
Figura 141: Muestra de los tres test en la misma gráfica [PMV][Opteron]

En estas gráficas finales podemos apreciar cómo se solapan ciertas zonas, pero conseguimos conocer la estructura de la función de fallos. Para ello hemos ejecutado tres Test con distintas configuraciones:

- Test1: Step size = 400 y comienzo en 1
- Test2: Step size = 400 y comienzo en 512
- Test3: Step size = 200 y comienzo en 512

Así podemos ver la gráfica completa, aunque cada ejecución requiere aproximadamente 3 horas de tiempo.

Lo mismo sucede en el Opteron para las ejecuciones con L2 algo diferente para las transferencias de bus:

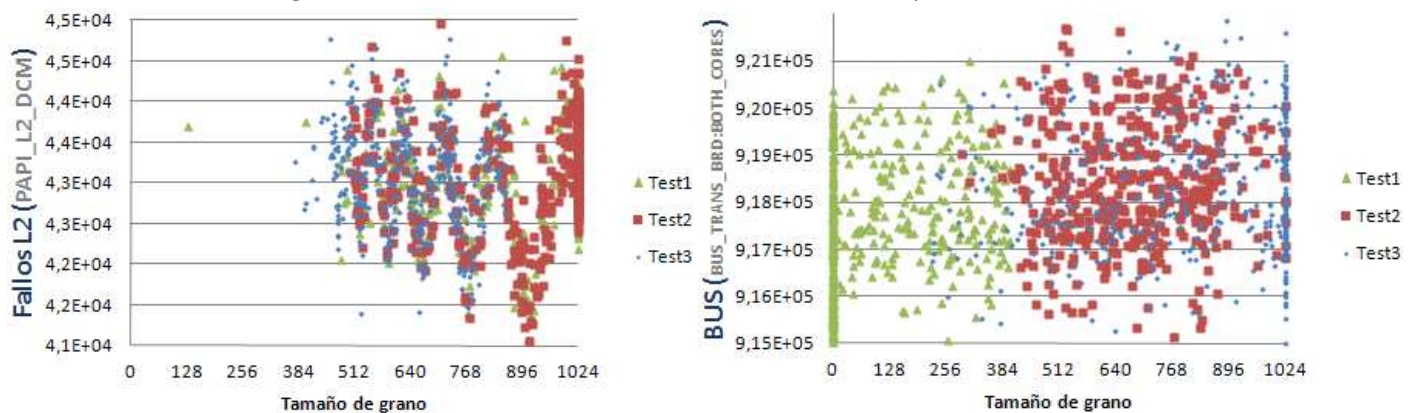


Figuras 142 y 143: Fallos en L2 y transferencias de bus con Simulated Annealing (3 tests) [PMV][Opteron]

El comportamiento de las gráficas es similar para los tamaños de grano sin embargo, volvemos a comprobar cómo *Simulated Annealing* ejecuta de distinta forma según la gráfica

que obtenemos y es por ello que deberíamos de seguir realizando pruebas aumentando el step size o comenzando en un tamaño de grano 850 para completar la parte derecha de la gráfica.

Sin embargo en el Core2Duo los resultados son mucho menos precisos:



Figuras 144 y 145: Fallos en L2 y transferencias de bus con Simulated Annealing (3 tests) [PMV][Core2Duo]

Los casos de fallos en L2 (Figura 144), también tenemos que la prueba test 1 no consigue centrarse en la parte izquierda de la función, a pesar de estar iniciando en 1.

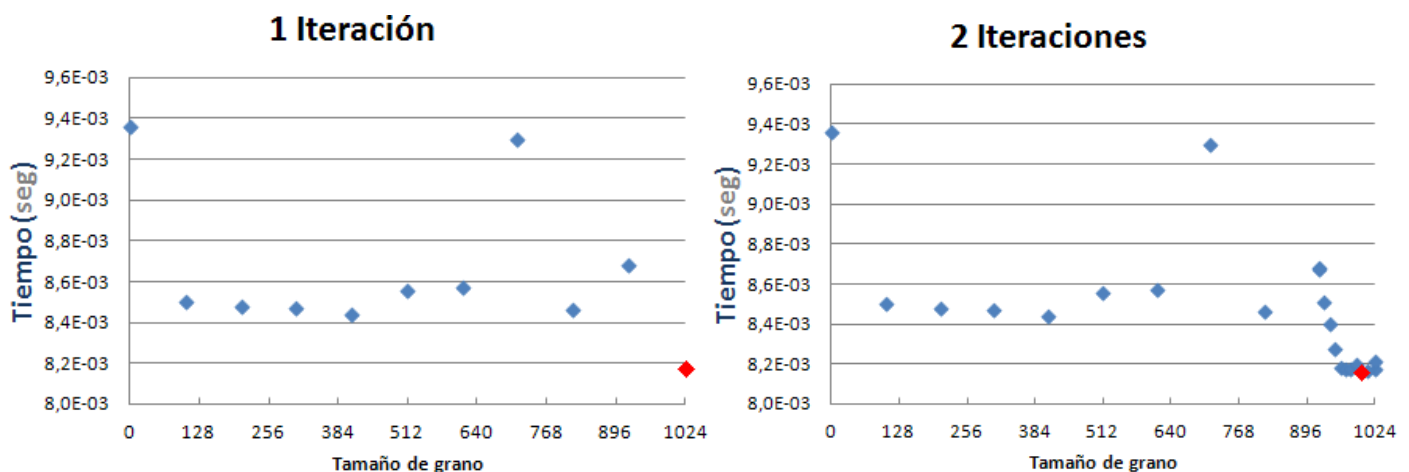
En el caso del bus, Figura 145, vemos una aleatoriedad casi total. Lo que nos dice esta gráfica es que no depende del tamaño de grano la saturación del bus y que en ocasiones consigue ahorrar muchas transferencias mientras que en otras la saturación es mayor. También tenemos que tener en cuenta que en este caso, los límites mínimos y máximo de la gráfica son muy pequeños y la diferencia de transferencias la cuantificamos en miles.

### 4.8.2 Búsqueda N-aria

De forma similar a como hicimos en *Simulated Annealing* vamos a tratar cada conjunto de 11 ejecuciones del benchmark como una iteración, una más que en el caso anterior, esto se debe a que dividiremos la gráfica en grupos de 10 (N) secciones. Cuando se delimiten y evalúen las secciones seleccionaremos las 3 (K) mejores para continuar con la exploración. Trabajamos sobre las mismas gráficas y con el mismo número de repeticiones por ejecución, con un valor de 5.

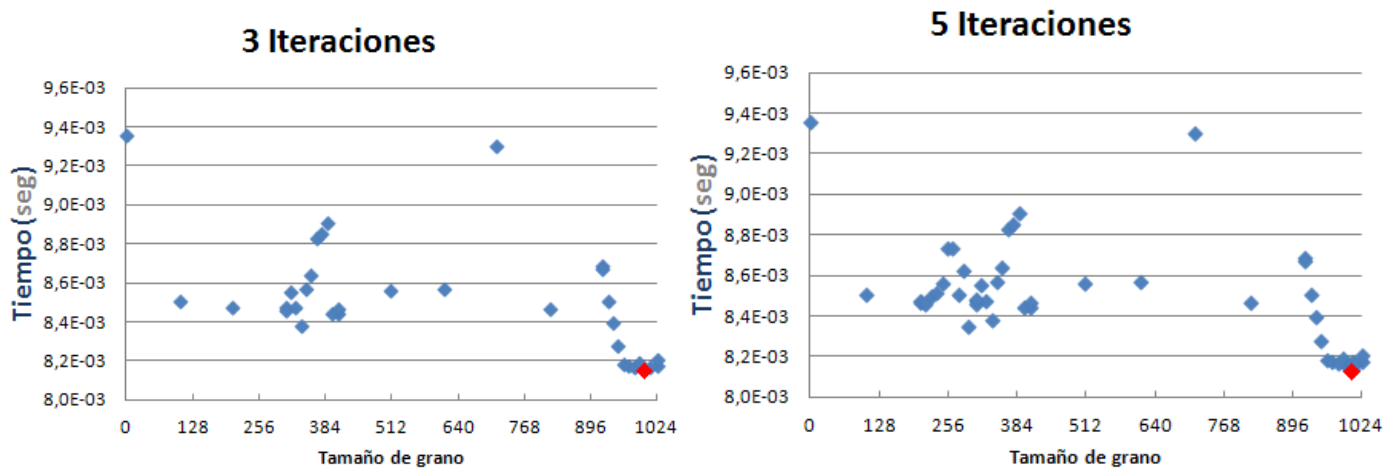
Al ser un algoritmo más sencillo en las gráficas vamos a ver más claramente cómo va ejecutando y por qué expande ciertas secciones y no otras. Este algoritmo tiene más restricciones a la hora de finalizar su ejecución. Como hemos visto *Simulated Annealing* puede seguir iterando hasta que alcance una temperatura mínima, sin importarle si alguna prueba la ha ejecutado ya previamente o no. La Búsqueda N-aria parará su ejecución cuando el tamaño de cada sección sea menor que N (10), o dicho de otra forma, cuando no podamos seguir dividiendo una nueva sección.

Las gráficas arrojadas por la Búsqueda N-aria son las siguientes:



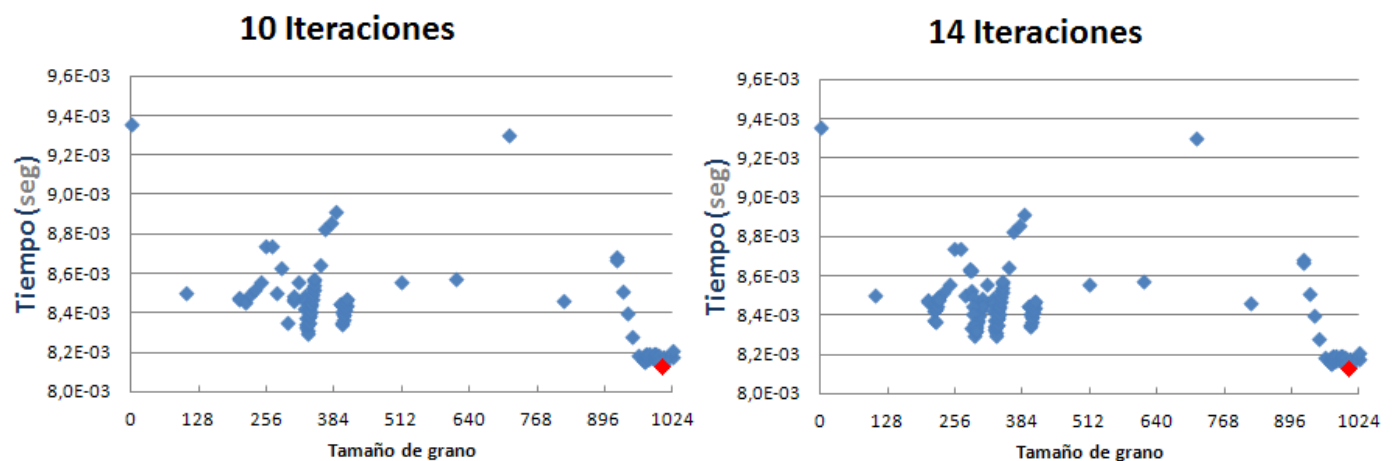
Figuras 146 y 147: Puntos testeados por la Búsqueda N-aria, 1 y 2 iteraciones [PMV][Opteron]

En la primera iteración vemos como se realiza la división en diez partes, y la mejor es la que se encuentra entre 919 y 1024 (Figura 146). Por tanto, en la segunda iteración es la sección que será explorada. En ella aparece un nuevo mínimo en el tamaño de grano 999 (Figura 147).



Figuras 148 y 149: Puntos testeados por la Búsqueda N-aria, 3 y 5 iteraciones [PMV][Opteron]

Para 3 y 5 iteraciones vemos como nos estamos centrando ahora en las zonas comprendidas entre 205 y 409. En la cuarta iteración volvemos a explorar una zona entre 909 y 1009 y se encuentra un nuevo mínimo en 1004.



Figuras 150 y 151: Puntos testeados por la Búsqueda N-aria, 10 y 14 iteraciones [PMV][Opteron]

Para finalizar aumentan las exploraciones en las secciones seleccionadas pero no aparecen nuevos mínimos. La catorceava es la última iteración ya que en este punto no pueden dividirse en más secciones las nuevas zonas a estudiar y la ejecución finaliza.

Si hacemos un estudio de los fallos en L1 tendremos lo siguiente:

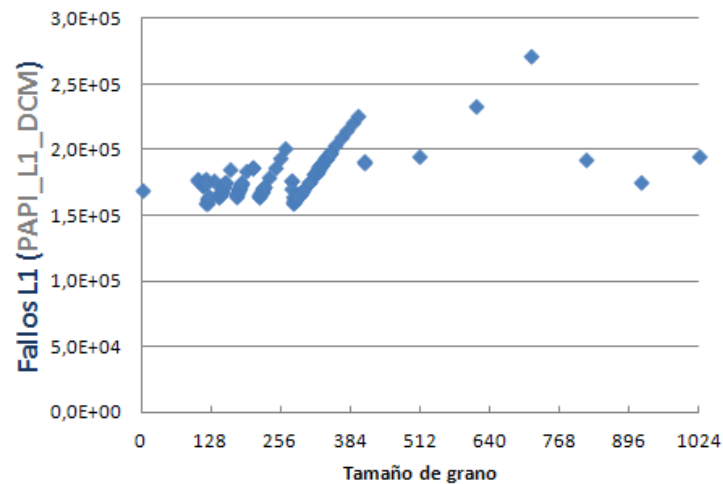


Figura 152: Fallos en L1 generados por la Búsqueda N-aria [PMV][Opteron]

Como hemos podido comprobar es razonable aumentar el número de zonas exploradas y de secciones para tener un mejor conocimiento de la función. En este apartado no seguiremos mostrando gráficas ya que los resultados obtenidos van a ser siempre los mismos por lo que pasaremos directamente al siguiente apartado en el que compararemos ambas técnicas de búsqueda de mínimos.

### 4.8.3 Comparativa

Para valorar los algoritmos de búsqueda de mínimos nos fijaremos principalmente en dos objetivos: 1º cuál es el mejor mínimo conseguido y 2º en qué iteración se consigue. En ambos casos según aumenta el número de ejecuciones disminuye la probabilidad de encontrar un mínimo global, es por ello que a partir de ciertas iteraciones no compensa la ejecución de más pruebas.

Como hemos visto en *Simulated Annealing* necesitamos unas 45 iteraciones (450 ejecuciones) para conocer aproximadamente la gráfica, a la búsqueda binaria le daremos por tanto las mismas oportunidades para encontrar el mínimo.

En el caso de la Búsqueda N-aria utilizaremos un  $K = 6$ , para de esta forma explorar un mayor número de zonas. Vamos a calcular hasta qué nivel de profundidad exploraremos:

1er Nivel  $\rightarrow$  11 Iteraciones (10 secciones)

2do Nivel  $\rightarrow 6 \cdot 11$  (6 exploradas) = 66 Iteraciones. Total arrastrado=77 Iteraciones

3er Nivel  $\rightarrow 6 \cdot 6 \cdot 11$  (36 exploradas) = 369 Iteraciones. Total arrastrado=446 Iteraciones.

4to Nivel  $\rightarrow 6 \cdot 6 \cdot 6 \cdot 11$  (216 exploradas) = 2376 Iteraciones. Total arrastrado = 2822 Iteraciones.

Por consiguiente, alcanzaremos levemente el nivel 4 de exploración. En el primer nivel la sección ocupa 1024, en el segundo nivel ocuparían en torno a 102 y en el tercer nivel 10. Luego vamos a tener una visión bastante detallada de la función.

Para comparar los resultados veremos los tres mejores resultados obtenidos, dentro de cada uno de ellos reflejaremos: el tamaño de grano con el que se consiguió ese resultado, el resultado en sí y la iteración en la que se encontró.

Estadística y mejor resultado		Técnica					
		Simulated Annealing			Búsqueda N-aria		
		Tamaño Grano	Resultado	Iteración	Tamaño Grano	Resultado	Iteración
<b>Mejor Tiempo (seg)</b>	1º	33	0.014923	63	693	0.014931	100
	2º	109	0.014936	53	715	0.014936	8
	3º	52	0.014943	45	103	0.015026	2
<b>Fallos L1</b>	1º	625	2269070.3	187	500	2269574.6	431
	2º	715	2272095.6	115	625	2270430.6	80
	3º	715	2272269.6	24	357	2271730.3	61
<b>Fallos L2</b>	1º	657	31015.3	31	683	37132.6	30
	2º	583	42057.3	21	715	37387.0	8
	3º	529	42119.3	10	511	43076.0	6
<b>Transferencias bus</b>	1º	879	840572.6	370	1008	846972.6	121
	2º	682	846289.0	137	897	885875.3	31
	3º	578	898959.0	107	1024	901505.0	11

Tabla 13: Resultados de Simulated Annealing y Búsqueda N-aria

Se pueden extraer muchas conclusiones de estos resultados la primera y principal es que *Simulated Annealing* es capaz de encontrar en todos los casos el mejor mínimo global, además lo consigue hacer siempre con un menor número de iteraciones. Excepto en el caso de las transferencias de bus que como dijimos es una gráfica totalmente aleatoria, aún así es capaz de encontrar un mejor mínimo global aunque requiere de 249 ejecuciones más.

Como contrapartida la Búsqueda N-aria consigue encontrar un segundo mínimo mucho antes que *Simulated Annealing*. Esto se debe principalmente a que la Búsqueda N-aria realiza un barrido general de la función mientras que *Simulated Annealing* se centra en algunas partes de la gráfica.

Los tamaños de grano que consiguen como mejores mínimos globales tienen cierta similitud. Ambos algoritmos a pesar de tener implementaciones muy diferentes y dispares acaban convergiendo a zonas de la gráfica similares.

Como resumen final vamos a hablar de las ventajas e inconvenientes de cada función.

#### **Simulated Annealing:**

Ventajas:

- 1.- Mejor funcionamiento general, encuentra el mejor mínimo global.

2.- El número de iteraciones para encontrar el mínimo global es menor.

Inconvenientes:

1.- Complejidad de la función.

2.- Puede dejar grandes zonas de la función sin explorar.

3.- Debemos variar mucho más los parámetros para conseguir una configuración que se adapte a la función. El punto de inicio y el tamaño de salto (STEP SIZE) dependen sobremanera de la función que queremos analizar.

4.- Problema de los límites, se deberían de ajustar para que no repita tanto el límite inferior ni superior. Es decir al devolver un tamaño de grano menor o igual a 1 siempre decimos que ejecute con 1. Se debería modificar para que pruebe con valores cercanos a 1, y no repita este valor, aunque estaríamos “engañando” al algoritmo ya que él quiere ejecutar con un tamaño de grano negativo y le estamos dando otro tipo de resultado.

5.- Repite iteraciones que ya ha ejecutado.

### **Búsqueda N-aria**

Ventajas:

1.- Simplicidad de la función.

2.- Encuentra buenos “segundos y terceros” mínimos globales, en un bajo número de iteraciones.

Inconvenientes:

1.- Puede dejar zonas sin analizar si tenemos los parámetros N y K bajos. En estas zonas ignoradas se podría encontrar el mejor mínimo global.

2.- Por esta misma razón este algoritmo funcionará mal para funciones con un gran número de pendientes. Sobre todo si estas son muy pronunciadas.

2.- Al igual que *Simulated Annealing* repite iteraciones.

Para solucionar la repetición de iteraciones se podrían ir almacenando los resultados para evitar así volver a ejecutar el benchmark, aunque como sabemos los resultados no siempre serán los mismos para un tamaño de grano, puesto que aunque son la misma prueba hay muchos factores que hacen que reproducir las mismas condiciones sea imposible. Este caso se puede observar en la tabla anterior, en la que *Simulated Annealing* encuentra dos mejores mínimos en la parte de fallos en L1. Ejecutando con el mismo tamaño de grano (715), los resultados aún siendo diferentes son bastante cercanos. Esta diferencia también se puede reducir aumentando el número de repeticiones del benchmark, las pruebas se realizaron bajo 5 repeticiones; aunque al aumentar el número de repeticiones supondría superar las 3 horas de ejecución por prueba.

## 5.- Comparativa Final

Como resumen final vamos a mostrar una comparativa general de las principales funciones y evaluaremos qué técnicas son más eficientes a la hora de ejecutar el benchmark.

Para hacer esta comparativa utilizaremos el tiempo como referencia. Realizaremos una normalización de los datos para que se pueda observar dentro de las diferencias existentes, en qué tamaños de grano se incrementan las diferencias. De tal modo que un 100% significaría que en ese tamaño de grano estamos viendo la mayor diferencia posible y un 0% nos indicaría que el tiempo de ejecución es similar teniendo en cuenta las diferencias existentes.

Número de filas	Tiempos escritura con stride (segundos)		
	Core2Duo	Opteron	Diferencia en %
1	2,744957	2,101881	02,816198 ↑
2	3,406234	1,438975	08,615141 ↑
4	4,449863	14,960122	46,027167 ↓
8	5,31293	15,392249	44,139969 ↓
16	13,686589	15,985286	10,066594 ↓
32	14,581863	21,163773	28,823902 ↓
64	14,843124	24,273878	41,299733 ↓
128	15,02612	24,19016	40,131723 ↓
256	14,911761	23,832408	39,065841 ↓
512	8,769154	22,901301	61,88836 ↓
1024	5,475513	22,319836	73,765687 ↓

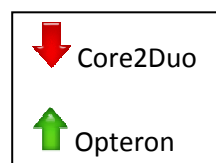


Tabla 14: Comparativa de tiempos escritura con stride entre Core2Duo y Opteron.

En la prueba con stride vemos la ganancia directa del Core2Duo. Como dijimos esta prueba está orientada a la ejecución de un solo hilo y, por tanto, estamos utilizando la mitad de los núcleos en el caso del Core2Duo y la doceava parte en el Opteron. Esta prueba demuestra que el Opteron queda totalmente limitado para pruebas monohilo. Además no sólo se ve limitado desde el punto de vista de núcleos también sucederá algo parecido en las memorias cachés. Tanto en L1 y L2 el Opteron sólo dispondrá de 1 de las 12 memorias cachés privadas que posee por núcleo para cada nivel, mientras que el Core2Duo tendrá una L1 privada y una L2 compartida.

Es curioso que los tiempos comiencen a subir antes en el Opteron, conforme aumentamos el número de filas, ya que la L1 del Opteron es de 64 KBytes y en el Core2Duo es de 32KB. La caché L1 del Opteron debería tardar más en llenarse y por tanto hacer mejor reutilización de datos. A pesar de esto, sí que es verdad que en acceso consecutivo funciona mejor el Opteron y es probable que se deba a tener una mayor capacidad de L1.



Tamaño de grano	Tiempos <i>false sharing</i> (segundos)		
	Core2Duo	Opteron	Diferencia en %
1	2,156685	4,771028	56,06517 ↓
2	3,978436	4,553111	12,324034 ↓
4	1,657337	1,376405	06,02465 ↑
8	1,565868	1,349419	04,641797 ↑
16	1,542872	1,305522	05,090024 ↑
32	1,501913	0,749863	16,12788 ↑
64	1,479857	0,571342	19,483307 ↑
128	1,443945	0,360358	23,237766 ↑
256	1,443877	0,221382	26,216679 ↑
512	1,399711	0,153977	26,715044 ↑
1024	1,436222	0,107985	28,48434 ↑

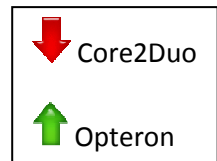


Tabla 15: Comparativa de tiempos *false sharing* entre Core2Duo y Opteron.

En esta segunda prueba el Opteron hace muestra de todo su potencial y mientras el Core2Duo ejecuta con únicamente dos hilos, el Opteron lo hace con 12. Con tamaños de grano pequeño vemos como se multiplica el efecto de *false sharing*. En el Core2Duo 2 hilos pelean por escribir en los mismos bloques de memoria, en el caso del Opteron son 12 hilos los que intentan acceder a un mismo bloque de memoria.

Cada bloque de datos ocupa 64 Bytes, y trabajamos con enteros de 4 Bytes, luego en cada bloque tendremos 16 elementos. Con un tamaño de grano 1, los 12 núcleos escribirían en un mismo bloque; con un tamaño de grano 2, 8 núcleos accederían al mismo bloque y así sucesivamente hasta que con un tamaño de grano 16 cada núcleo podría llegar a tener (si los datos se encuentran alineados con el comienzo de los bloques) sus bloques de forma independiente.

En cuanto a los tamaños de grano superiores el Opteron funciona mejor, a partir de un tamaño de grano 126 el Opteron sextuplica la capacidad de procesamiento del Core2Duo. Con esto se demuestra que el Opteron a máximo rendimiento es capaz de resolver problemas de procesamiento mucho más rápido, siempre que no existan dependencias de datos que hagan que varios núcleos compartan bloques de memoria.

	Tiempos <b>producto matriz-dispersa vector</b> (segundos)		
Tamaño de grano	Core2Duo	Opteron	Diferencia en %
1	0,015084	0,009415	82,040521 ↑
2	0,014968	0,009054	85,586107 ↑
4	0,014846	0,008722	88,625181 ↑
8	0,014787	0,00867	88,523878 ↑
16	0,014801	0,008558	90,347323 ↑
32	0,014801	0,008518	90,926194 ↑
64	0,014799	0,008511	90,998553 ↑
128	0,014777	0,008441	91,693198 ↑
256	0,014933	0,008756	89,392185 ↑
512	0,014921	0,008553	92,156295 ↑
1024	0,014902	0,008174	97,366136 ↑

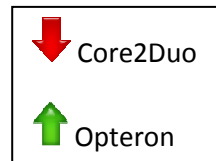


Tabla 16: Comparativa de tiempos producto matriz vector entre Core2Duo y Opteron.

En la prueba de producto matriz-dispersa vector, en la que ambas plataformas funcionan a máximo rendimiento cabe destacar que el Opteron consigue mejores tiempos de forma clara y con una gran ventaja.

Si en las pruebas monohilos vimos que su capacidad está claramente limitada, con pruebas de computación a gran escala funciona mucho mejor. Esta arquitectura ha sido diseñada para realizar grandes labores de computación y queda demostrado en esta prueba, que es la que más exige en cuanto a carga de datos. También es muy importante el uso de la L3 compartida, en este caso hay dos memorias de 6 MBytes a compartir en dos grupos de 6 núcleos. En el Core2Duo sólo tenemos L2 compartida de 2 MBytes entre dos núcleos.

De nuevo tenemos un efecto parecido a *false sharing*. Cuanto más se aumenta el tamaño de grano más reducimos alguna posible dependencia que podría existir, este caso en el vector de escritura del resultado del producto.

En la parte de Huge Pages las diferencias son las siguientes:

Tamaño de grano	Tiempos <i>false sharing</i> (segundos)			
	Huge Pages Off	Huge Pages On	Diferencia en %	
1	2,156685	2,147051	0,00373595	↑
2	3,978436	3,790767	0,07277589	↑
4	1,657337	1,678848	0,00834172	↓
8	1,565868	1,629234	0,02457261	↓
16	1,542872	1,602982	0,02330997	↓
32	1,501913	1,587849	0,033325	↓
64	1,479857	1,574157	0,03656846	↓
128	1,443945	1,568203	0,04818583	↓
256	1,443877	1,560468	0,04521265	↓
512	1,399711	1,572278	0,0669195	↓
1024	1,436222	1,583441	0,05708984	↓

↓ HP Off  
↑ HP On

Tabla 17: Comparativa de tiempos en *false sharing* con/sin Huge Pages [Core2Duo]

Las diferencias entre activar o no Huge Pages son mínimas pero los resultados no favorecen a la activación de esta técnica, ya que en la mayoría de los tamaños de grano tenemos un aumento del tiempo.

En cuanto a las técnicas de optimización podríamos decir que ambas son bastante interesantes aunque si nos tuviésemos que decantar por una sería *Simulated Annealing*, que ha conseguido demostrar ser la que encuentra mejor mínimos globales en un menor número de iteraciones. Sin embargo, por la simplicidad de la Búsqueda N-aria y porque funciona mejor con gráficas más aleatorias recomendamos no olvidar esta técnica.

También cabe destacar que deberemos modificar las opciones de comienzo y tamaño de salto de *Simulated Annealing* para tratar de cubrir todas las posiciones de la gráfica, y que ésta no se centre en ciertas zonas donde no esté el mínimo global. Por su parte, para la Búsqueda N-aria parece que un N=10 (número de secciones en las que dividimos la gráfica o sección) y K=6 (Número de secciones a explorar), son una buena configuración a la hora de buscar mínimos.

## 6.- Planificación y presupuesto

En este apartado se aborda todo lo concerniente al presupuesto y la planificación del proyecto. Para tratar estos aspectos comenzaremos explicando cómo se fue desarrollando el proyecto, en la parte final se muestra el presupuesto.

### 6.1 Planificación

La idea comenzó como la creación de un trabajo dirigido más proyecto fin de carrera, estando ambos directamente relacionados. Pensábamos en realizar un estudio inicial de familiarización con la librería PAPI, capaz de leer contadores hardware y realizar pruebas pequeñas. Este trabajo dirigido se adjunta en el proyecto como apéndice. El siguiente paso, después de finalizar este trabajo dirigido fue el utilizar estas herramientas para implementar el benchmark.

Este proyecto es complejo desde el punto de vista de planificación, al ser una aplicación destinada a un cliente con ciertas exigencias desde el punto de vista técnico. Además no podíamos dar por hecho la utilización de ciertas librerías, puesto que algunas de ellas no se pueden ejecutar dependiendo del sistema operativo o de la arquitectura que estemos utilizando. Esta desventaja suponía suficiente incertidumbre como para no poder fijar puntos en el horizonte como hitos durante el desarrollo del proyecto.

Las tres etapas principales en las que podemos diferenciar el proyecto son:

- Trabajo dirigido: Etapa en la que comenzamos a familiarizarnos con los contadores PAPI y su funcionamiento.
- Implementación del Benchmark: la etapa más larga y costosa, en la que se desarrolla la parte principal del benchmark, así como se agregan mejoras en el código inicial del trabajo dirigido.
- Documentación: parte final, que se encarga de detallar y explicar la motivación y objetivos del proyecto.

El proyecto no se desarrolló de forma continuada si no que tuvo dos parones, uno debido al gran cúmulo de asignaturas pendientes durante el cuatrimestre y otro debido a la realización de una beca en la empresa AXA Seguros.

En la siguiente figura veremos el diagrama con las fases principales del proyecto y las 30 tutorías que fueron necesarias concertar con el tutor de este proyecto: David Expósito Singh. La duración media de las tutorías fue aproximadamente de 45-50 minutos. El resto de asuntos se resolvió mediante correo electrónico y hasta la fecha de redacción de esta memoria fueron necesarios 205 emails. También fue necesaria una tutoría con el profesor Francisco Javier García Blas para la introducción a la VPN de la universidad así como a la utilización del AMD Opteron.

## 6.- Planificación y presupuesto

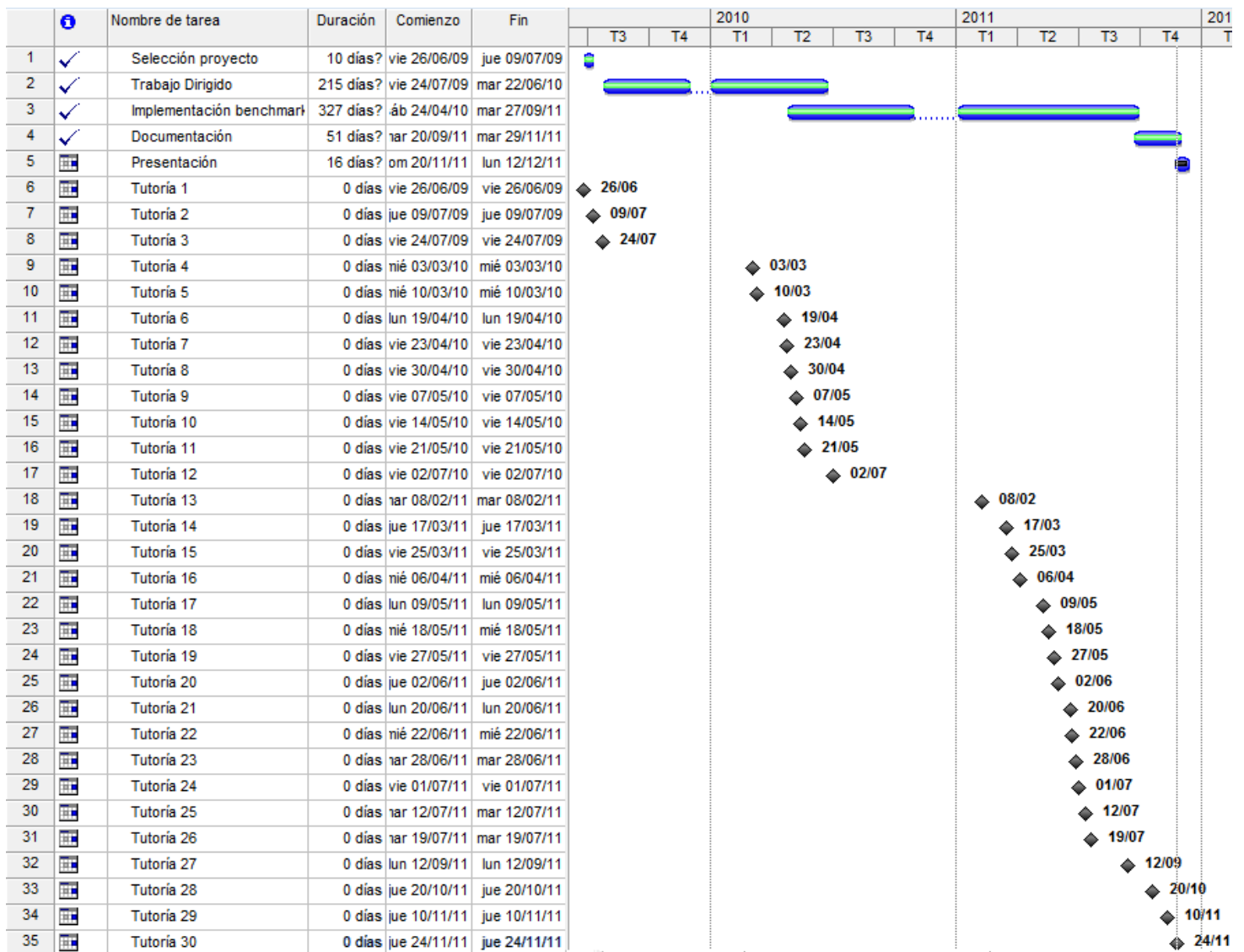


Figura 153: Fases del desarrollo del proyecto junto con las tutorías

En la imagen se observan claramente las dos veces en las que el proyecto quedó relegado a un segundo plano y como durante estos períodos las tutorías fueron prácticamente nulas. Los momentos de mayor dificultad a la hora de desarrollar probablemente queden mejor identificados cuando la frecuencia de tutorías es mayor.

Sobre el benchmark llevamos un control de versiones, que eran comprimidas en archivos y enviadas al correo. En total fueron realizadas 37 versiones para alcanzar la versión 1.0 y final, que es la que se adjunta junto con esta memoria y la presentación.

En la figura que tenemos a continuación veremos la creación de estas versiones, en función de la línea del tiempo.

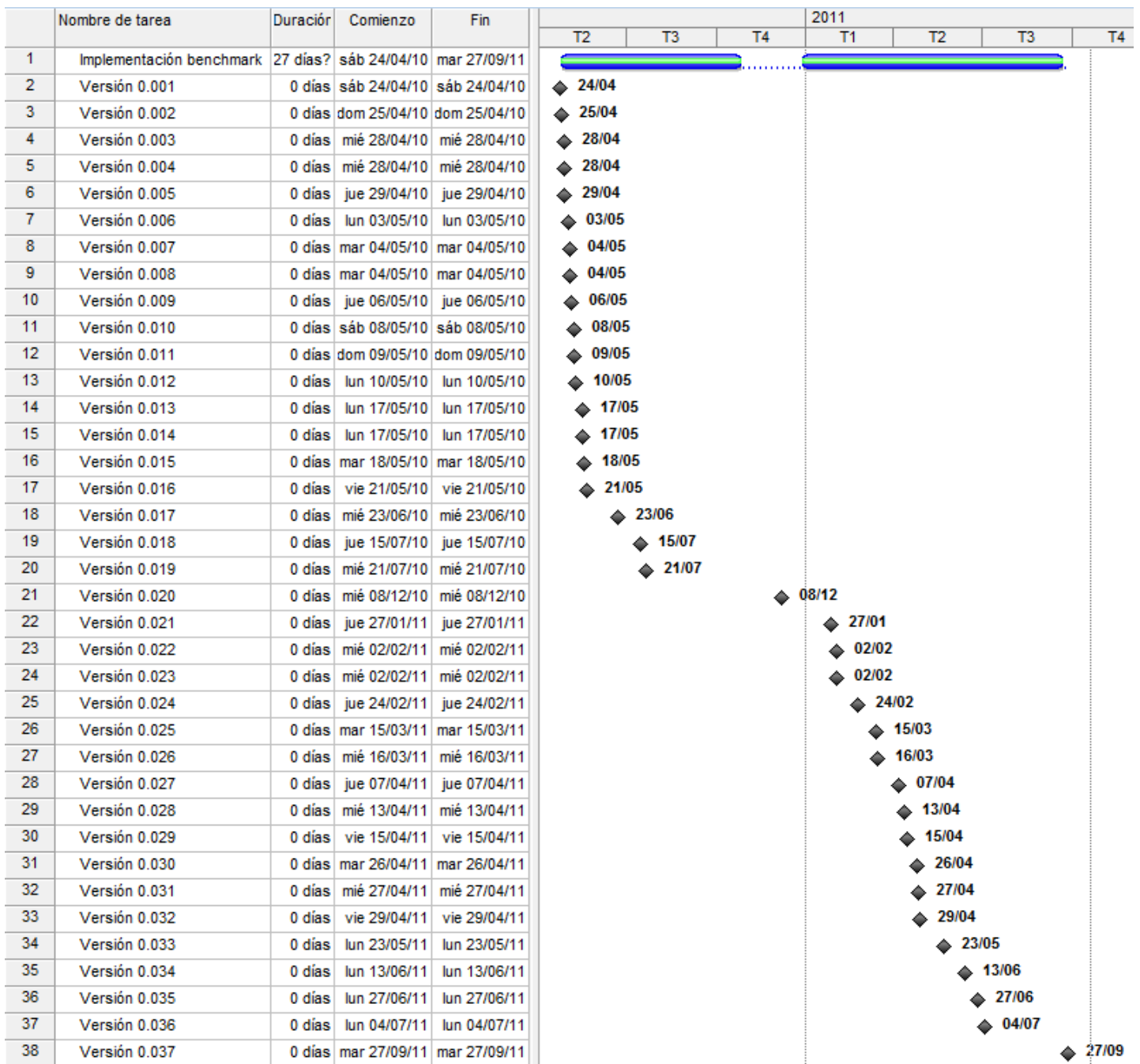


Figura 154: Diagrama de versiones durante la fase de implementación

La vista de la implementación refleja la necesidad de un gran número de versiones al comienzo del desarrollo. Conforme cerrábamos partes del proyecto comenzaban a tener más peso las pruebas y la experimentación y los cambios añadidos cada vez eran menores.

Para tener un punto de vista aproximado de la carga de trabajo por las distintas temáticas del proyecto en el siguiente diagrama mostramos la cantidad de líneas de código generadas para cada parte:

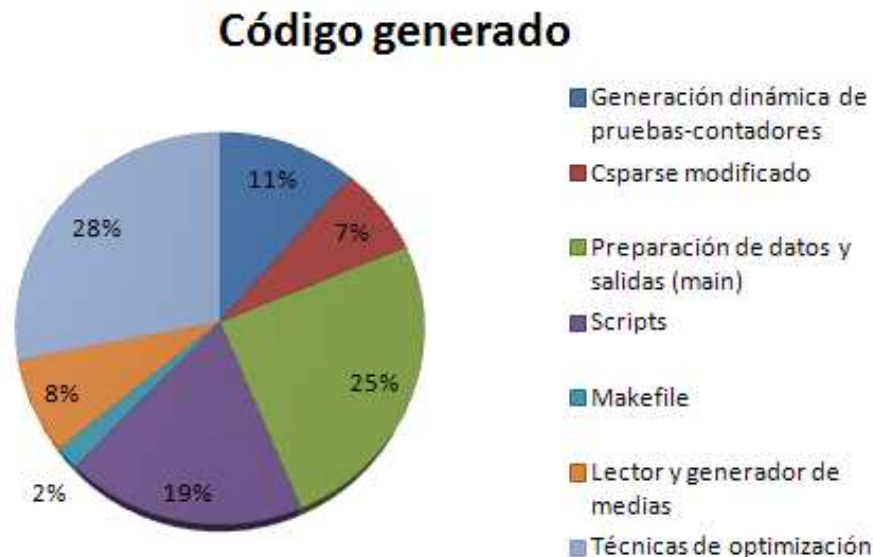


Figura 155: Diagrama de código generado

Las técnicas de optimización (*Simulated Annealing* y *Búsqueda N-aria*), son las que más código han requerido. Después de esto la preparación de datos y salidas, el método main, es el más costoso en cuanto a líneas de código. Los terceros a la cabeza serían los distintos scripts para generar pruebas para distintos parámetros (número de filas, tamaño de grano) así como para generar estadísticas (media y desviación típica). En la cuarta posición estaría la generación de pruebas-contadores, es decir, la parte en la que con un conjunto de contadores de entrada obtenemos las distintas ejecuciones a realizar con los contadores agrupados. En el antepenúltimo puesto estaría el lector y generador de medias. En rojo vemos la parte del producto matriz-dispersa vector que añadimos para poder realizar la multiplicación por filas, evitando así las redundancias. Y para acabar tendríamos el Makefile encargado de compilar y generar todos los ejecutables.

También hemos generado una TagCloud <sup>[59]</sup>, o nube de términos para conocer cuáles son las palabras que más aparecen en este documento. Se muestran ordenadas alfabéticamente siendo las palabras de mayor tamaño las que más aparecen:

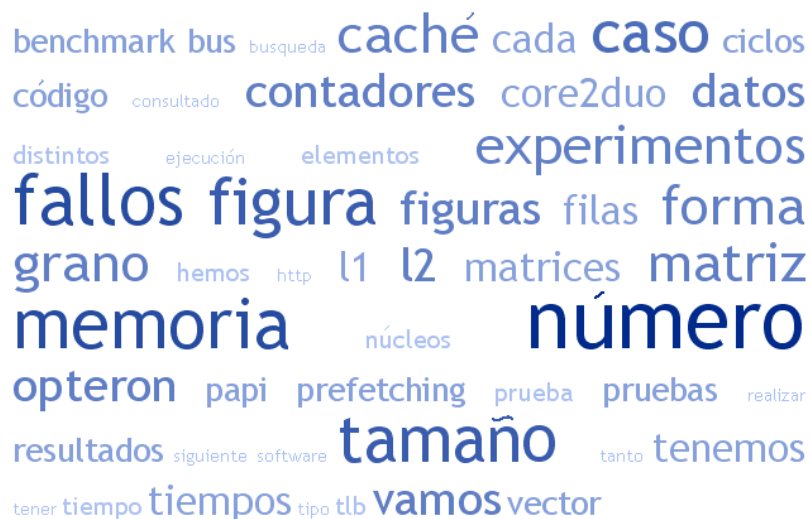


Figura 156: TagCloud realizado sobre la memoria



## 6.2 Presupuesto

En cuanto al número de horas de trabajo han sido aproximadamente de 2 horas diarias durante 26 meses. Cabe aclarar que estas dos horas no han sido continuas durante todos los días laborables y que se ha trabajado también en días no laborables. Por ser el primer trabajo estimamos unos 15€/hora. Si calculamos el total de horas en hombre mes (131,25), estaríamos hablando de 9,14. El tutor del proyecto trabajó como consultor e investigador y por ello se le estiman 35€ por hora durante un mes.

En los costes de equipos incluimos las dos máquinas cuyas arquitecturas fueron evaluadas del proyecto. Uno de ellos es propiedad del programador y el otro fue cedido de parte de la Universidad Carlos III. Para tener una idea del coste del Opteron, lo plantearemos como si hubiésemos solicitado un servicio de computación a una empresa. Sabemos que el precio en estos casos se ofrece por horas, y si hablásemos de un supercomputador como el Finis Terrae del Centro de Supercomputación de Galicia estamos hablando de un coste de 20 céntimos la hora <sup>[60]</sup>. Como el Opteron es un ordenador mucho más simple vamos a suponer un coste de 8 céntimos la hora. El tiempo total de ejecución en el Opteron rondó las 60 horas.

En cuanto a los gastos directos incluimos el transporte necesario durante las 30 tutorías tanto en la ida como en la vuelta, y suponemos el coste de un viaje de 80 céntimos de euro (utilizando bonos de 10 transportes). También se añade el coste de la impresión y encuadernación del proyecto así como otros gastos necesarios para el desarrollo del proyecto.

El coste total del benchmark es de 27.930,74 € y se justifica desglosado en el presupuesto de la siguiente página. Para sopesar los costes con el estado actual de los benchmarks que no son libres o son *freeware*, vamos a comparar el precio de un benchmark con el coste para desarrollar el AdaptBenchmark.

Una de las empresas más conocidas en la materia de creación de benchmarks es SPEC. SPEC hace diferencia entre si el producto será vendido a una organización sin ánimo de lucro, en la cual el coste de la venta se verá reducido a la mitad; o en el caso en que la venta sea para un particular (*retail*) valdrá el doble de su precio <sup>[61]</sup>. Si tomamos el benchmark más reciente que han creado para CPU, el SPEC 2006 el coste de actualización a éste es de 400€ y la compra como particular supone un gasto de 800€ <sup>[62]</sup>. Por tanto, a este coste tendríamos que vender un total de 35 unidades (27.930,74/800) para obtener algún tipo de beneficio.

También somos conscientes de que nuestro benchmark no posee tantas funcionalidades como el SPEC 2006. Si impusiésemos una estimación de venta de 700 unidades podríamos rebajar el coste del AdaptBenchmark a 40 € (27.930,74/700), algo que sería mucho más económico y acorde al producto que estamos ofreciendo.





## UNIVERSIDAD CARLOS III DE MADRID

### Escuela Politécnica Superior

#### PRESUPUESTO DE PROYECTO

**1.- Autor:** Antonio Díaz Ponce

**2.- Departamento:** Informática, Escuela Politécnica Superior

**3.- Descripción del Proyecto:**

- Título **AdaptBenchmark: El benchmark Adaptativo Universal**  
 - Duración (meses) **26**  
 Tasa de costes Indirectos: **20%**

**4.- Presupuesto total del Proyecto (valores en Euros):**

Euros **27.930,74 €**

**5.- Desglose presupuestario (costes directos)**

**PERSONAL**

Apellidos y nombre	Categoría	Dedicación (hombres mes) <sup>a)</sup>	Coste hombre mes	Coste (Euro)	Firma
Antonio Díaz Ponce	Programador	9.14	1.968,75 €	17.994,37 €	<i>Antonio Díaz</i>
David Expósito Singh	Investigador/consultor	1	4.593,75 €	4.593,75 €	<i>David Singh</i>
		<b>Hombres mes</b>	<b>10.14</b>	<b>Total</b>	<b>22.588,12 €</b>

<sup>a)</sup> 1 Hombre mes = 131,25 horas. Máximo anual de dedicación de 12 hombres mes (1575 horas)  
 Máximo anual para PDI de la Universidad Carlos III de Madrid de 8,8 hombres mes (1.155 horas)

**EQUIPOS**

Descripción	Coste (Euro)	% Uso dedicado proyecto	Dedicación (meses)	Periodo de depreciación	Coste imputable <sup>d)</sup>
Intel Core2Duo	775,00	50	24	96 meses	96,88 €
AMD Opteron	30 * 0,08	100	2	1 mes	4,80 €
<b>Total</b>					<b>101,68 €</b>

<sup>d)</sup> Fórmula de cálculo de la Amortización:

$$\frac{A}{B} \times C \times D$$

A = nº de meses desde la fecha de facturación en que el equipo es utilizado

B = periodo de depreciación (60 meses)

C = coste del equipo (sin IVA)

D = % del uso que se dedica al proyecto (habitualmente 100%)

**SUBCONTRATACIÓN DE TAREAS**

No fue necesaria la subcontratación de tareas para la realización de este proyecto

**OTROS COSTES DIRECTOS DEL PROYECTO<sup>e)</sup>**

Descripción	Empresa	Costes imputable
Luz	Iberdrola	120,00 €
Internet	Telefónica	85,10 €
Transporte	CRTM	48,00 €
Maletín portátil	Informática T70	30,00 €
Papel Impresora	Papelería Copicentro	60,00 €
Encuadernación	Papelería Copicentro	63,25 €
Consumibles	Papelería Copicentro	180,00 €
<b>Total</b>		<b>586,35 €</b>

<sup>e)</sup> Este capítulo de gastos incluye todos los gastos no contemplados en los conceptos anteriores, por ejemplo: fungible, viajes y dietas, otros,...

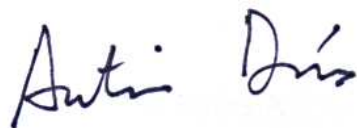
**6.- Resumen de costes**

Presupuesto Costes Totales	<b>Presupuesto Costes Totales</b>
Personal	22.588,12 €
Amortización	101,68 €
Subcontratación de tareas	0 €
Costes de funcionamiento	586,35 €
Costes Indirectos	4.654,59 €
<b>Total</b>	<b>27.930,74 €</b>

El presupuesto total de este proyecto asciende a la cantidad de VEINTISIETE MIL NOVECIENTOS TREINTA CON SETENTA Y CUATRO EUROS (27.930,74 €).

Leganés a 20 de Noviembre de 2011

El ingeniero proyectista



Fdo. Antonio Díaz Ponce

## 7.- Conclusiones

---

La realización de este proyecto ha supuesto todo un reto, ya que el desarrollador cursó como especialidad Inteligencia Artificial, la cual no estaba muy relacionada con el tema del proyecto. Además durante la carrera no se trata mucho el tema de funcionamiento de benchmarks, aunque sí se asientan bien las bases sobre el funcionamiento de la arquitectura básica.

El benchmark al principio suponía todo un reto, puesto que el desconocimiento de librerías era prácticamente total, excepto OpenMP que utilizamos en Arquitectura de Computadores II, quizás la asignatura que tenga una relación más directa con este tema. Personalmente la mayor dificultad fue encontrada con las distintas librerías que íbamos incluyendo, y sobre todo porque en alguna de ellas tuvimos que ver el código, entenderlo y modificarlo, como fue el caso del producto matriz-dispersa vector (CSparse).

Otra de las grandes dificultades es el tiempo de espera. En este tipo de proyectos es aconsejable tener varios frentes abiertos ya que no es recomendable esperar a que finalicen las pruebas para seguir trabajando. En algunas ocasiones dentro de una gráfica hemos llegado a representar 9 horas de ejecución correspondientes a tres tests. Dentro de alguna página de la memoria puede haber hasta 24 horas de ejecución, tiempo que no podemos estar parados esperando para tomar los datos, prepararlos en las gráficas y comentar; debemos tener pensado algún otro tipo de trabajo que se pueda realizar de forma paralela a las pruebas.

Esperamos que con esta memoria quede claro el funcionamiento y la estructura del benchmark puesto que es un tema algo complejo. Para ello hemos intentado incluir el mayor número de figuras y gráficas explicativas que mejoran, a nuestro punto de ver, el entendimiento de este proyecto de fin de carrera.

Estamos contentos con los resultados obtenidos, aunque bien es cierto que algunos de ellos no hemos conseguido explicar al 100%. Tenemos que tener en cuenta que en este proyecto no sólo ofrecemos un benchmark si no que detrás de éste hay un estudio de experimentación sobre el funcionamiento de dos arquitecturas.

A modo de resumen vemos que los objetivos marcados inicialmente se han cumplido:

- Hemos utilizado nuevas tecnologías para desarrollar el benchmark y tomar estadísticas del microprocesador, como PAPI y OpenMP. Además hemos aprendido a utilizar otras técnicas relacionadas con el análisis del rendimiento de un microprocesador, como *Huge Pages* y *Prefetching*.
- Desarrollamos una nueva técnica de búsqueda de mínimos, que hemos denominado Búsqueda N-aria, que ha sido muy útil a la hora de ser comparada con *Simulated Annealing*.
- Realizamos una metodología de empleo y acceso a los contadores para asegurarnos que los resultados obtenidos sean lo más exactos posibles, mediante técnicas como borrado de caché, o obtención de medias aritméticas y desviaciones típicas. También hemos conseguido que este benchmark sea capaz de tomar

distintos contadores y ejecutar, tanto si éstos están disponibles para dicha arquitectura o no (en el último caso son ignorados). Mediante la implementación de la multiplexación de forma manual conseguimos tomar estadísticas de más contadores de los que nos permiten los registros de la arquitectura.

- Comparamos ambas arquitecturas multicore (Intel Core2Duo y AMD Opteron) y fue exitosa la búsqueda de un contexto en el que ambas arquitecturas ejecutan distintas pruebas en el menor tiempo posible.

La mayoría de referencias que hemos encontrado han sido Online y es por ello que el Apartado 8 está formado casi principalmente por enlaces. Cada vez más esta información se encuentra en dispositivos digitales. Además la información en ocasiones es difícil de encontrar ya que son temas que en ocasiones no se publican. Esto se debe a que son competencias directas de las marcas principales creadoras de microprocesadores y, un exceso de información, puede resultar muy útil para la competencia.

## 7.1 Líneas futuras

Las posibilidades de ampliar los estudios son muy variadas y hay un gran número de líneas futuras sobre las que se podría trabajar para mejorar el benchmark.

Una de ellas sería tomar estadísticas en todos los hilos. De esta forma podríamos conocer mejor lo que está ocurriendo en cada uno de los hilos <sup>[63]</sup>. Para ello deberíamos crear, inicializar, arrancar y parar los contadores para cada proceso. Será necesario utilizar “bound threads”, se denomina así a los procesos que ejecutan en el contexto de un proceso de peso ligero, en los que ningún otro hilo puede ejecutar <sup>[64]</sup>. Además en esta mejora sería interesante tomar estadísticas durante la ejecución de éste para conocer la saturación en función del tiempo, es decir, tomar estadísticas durante la ejecución del benchmark. Sin embargo, esto último afectaría a los tiempos por lo que habría que adoptar una solución de compromiso.

Para mostrar los datos en tiempo real se podría utilizar Perfometer <sup>[65]</sup>. Perfometer es una librería asociada a PAPI capaz de mostrar los resultados de los contadores en gráficas. Está programado en Java por lo que el único escollo que podríamos tener a la hora de implementar este software sería variar nuestro código para mandar los datos mediante *sockets*. En la Figura 157 podemos ver una captura de Perfometer:

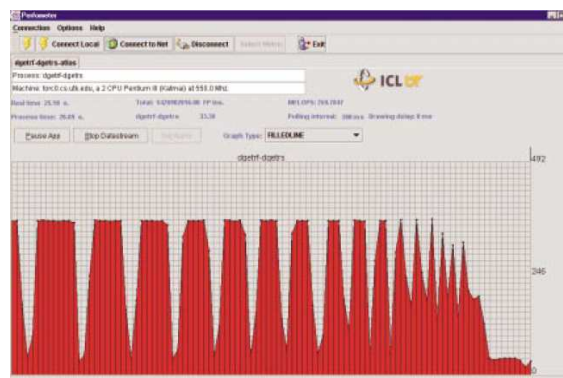


Figura 157: Captura de pantalla de la aplicación Perfometer

Otra de las posibles mejoras sería en la parte de selección de conjuntos de contadores, ahora lo que hacemos es seguir una lista de contadores y ver con cuantos podemos ejecutar en una misma prueba, siguiendo el orden que nos da el fichero de texto plano. Sin embargo, se podría mejorar esto para que se minimizase el número de pruebas y así conseguir reducir los tiempos de ejecución por causas de la multiplexación.

También podrían ser interesantes las lecturas de otro tipo de contadores secundarios y centrarnos en otras partes menos principales, como número de operaciones en coma flotante o algunos de los contadores que vimos en la Sección 2.3 Contadores hardware.

Además se podrían implementar distintas pruebas para intentar testear diferentes aspectos de la arquitectura o para intentar comprobar cómo funciona con distintos tipos de código, como recursividad o probar más funciones con matrices dispersas (factorizaciones).

En cuanto a las técnicas de optimización podríamos mejorar ambos casos para no repetir ejecuciones ya realizadas e ir almacenando los resultados producidos por el benchmark para los casos en los que los algoritmos deseen obtener una salida. Además en el caso de *Simulated Annealing* podríamos hacer que si se desea explorar un tamaño de grano menor que uno, calculemos un rebote mediante una función aleatoria para no repetir tantas veces las pruebas sobre los límites 1 y 1024 (límites establecidos como mínimo y máximo).

Como última posible línea futura del proyecto sería posible ejecutar el benchmark en un mayor número de arquitecturas, y establecer alguna especie de nota final según los resultados obtenidos en cada prueba. Para obtener esta “nota” final podríamos dar distintos pesos a cada una de las pruebas y mediante la suma de estos valores obtendríamos la nota final.

## 8.- Glosario

---

Glosario de términos cuyos significados son imprescindible para la comprensión del proyecto:

**Arquitectura** → Estructura lógica y física de los componentes de un computador.

**Benchmark** → Proceso encargado de comparar productos o servicios.

**Bus** → Sistema digital que transfiere datos entre los componentes de una computadora, o entre distintos computadores.

**Caché** → Sistema de almacenamiento de alta velocidad integrado en el microprocesador.

**Ciclo** → Es el período que tarda la CPU en ejecutar una instrucción de lenguaje máquina.

**CPU** → Unidad Central de Proceso, componente del computador que interpreta las instrucciones contenidas en los programas y procesa los datos.

**Compilador** → Programa informático que traduce un programa escrito en un lenguaje de programación a otro lenguaje de programación, normalmente código máquina.

**Contador hardware** → Estadística o valor que se toma sobre un aspecto físico de la arquitectura.

**Fallo caché** → Situación que se produce cuando un dato solicitado por el microprocesador no se encuentra en la memoria caché.

**Hilo** → es la unidad de procesamiento más pequeña que puede ser planificada por un sistema operativo.

**Instrucciones** → conjunto de datos insertados en una secuencia estructurada que el procesador interpreta y ejecuta.

**Kernel** → Parte principal del sistema operativo.

**Librería** → Conjunto de subprogramas utilizados para desarrollar software.

**Memoria principal** → Dispositivo interno de almacenamiento de datos e instrucciones del ordenador. Se comunica con el microprocesador de la CPU mediante el bus de direcciones.

**Multiplexación** → Acción consistente en ejecutar una misma prueba con distintos contadores hardware.

**Núcleo (core)** → Parte más reducida del ordenador con capacidad de proceso.

**Prefetching** → Técnica que consiste en precargar en memoria caché bloques de memoria que se prevé que serán cargados.

**Procesador (micro)** → Circuito constituido por millares de transistores integrados en un chip, que realiza alguna determinada función de los computadores electrónicos digitales.

**Salida** → Resultados que muestra una función o programa.

**Script** → Programa usualmente simple, que por lo regular se almacena en un archivo de texto plano.

**Stride** → Salto, zancada.

**Tamaño de grano (chunk)** → Es la unidad mínima de reparto que se asigna a cada núcleo.

## 9.- Referencias

- [1]. ¿Qué es el benchmarking? Consultado el [23/09/2011]  
<http://www.e-conomic.es/programa/glosario/definicion-de-benchmarking>
- [2]. Wordreference Consultado el [23/09/2011]  
<http://www.wordreference.com/es/translation.asp?tranword=benchmark>
- [3]. PassMark Software Consultado el [24/09/2011]  
<http://www.cpubenchmark.net/>
- [4]. Linkpack Consultado el [24/09/2011]  
<http://www.netlib.org/linpack/>
- [5]. NAS Pararell Benchmarks Consultado el [24/09/2011]  
<http://www.nas.nasa.gov/Resources/Software/npb.html>
- [6]. SPEC Consultado el [24/09/2011]  
<http://www.spec.org/spec/>
- [7]. Futuremark Consultado el [24/09/2011]  
<http://www.futuremark.com/>
- [8]. ICOMP Consultado el [24/09/2011]  
<http://developer.intel.com/design/pentiumii/perfbref/243393.htm>
- [9]. Listado de benchmarks en Fortran Consultado el [24/09/2011]  
<http://www.personal.psu.edu/hdk/fortran.html#Benchmarks>
- [10]. Características técnicas Toshiba Satellite A200-1dy Consultado el [27/09/2011]  
[http://static.compusa.com/pdf/Toshiba\\_Satellite\\_A200\\_SatellitePro\\_A200\\_Series-UserManual.pdf](http://static.compusa.com/pdf/Toshiba_Satellite_A200_SatellitePro_A200_Series-UserManual.pdf)
- [11]. SLA4A (Intel Core 2 Duo) Consultado el [27/09/2011]  
<http://www.cpu-world.com/sspec/SL/SLA4A.html>
- [12]. Optimizing Embedded System Performance - Impact of Data Prefetching on a Medical Imaging Aplication Consultado el [28/09/2011]  
[ftp://download.intel.com/technology/advanced\\_comm/315256.pdf](ftp://download.intel.com/technology/advanced_comm/315256.pdf)
- [13]. Intel Core 2 Duo – Test Consultado el [28/09/2011]  
<http://www.behardware.com/articles/623-6/intel-core-2-duo-test.html>
- [14]. AMD Opteron 6168 Consultado el [28/09/2011]  
<http://www.cpu-world.com/CPU%20K10/AMD-Opteron%206168%20-%20OS6168WKTCEGO%20%28OS6168WKTCEGOWOF%29.html>
- [15]. WR Cluster Hardware Consultado el [28/09/2011]  
<http://wr0.wr.inf.h-brs.de/wr/hardware/hardware.html>
- [16]. Server Processors: Why More Cores Matter Consultado el [29/09/2011]  
<http://sites.amd.com/us/business/promo/server/Pages/why-amd-for-business.aspx#3>
- [17]. IBM supercomputer more powerful than 1.5-mile-high stack of laptops Consultado el [29/09/2011]  
<http://www.networkworld.com/news/2007/062607-ibm-supercomputer.html>
- [18]. Sourceforge.net perfom2 files Consultado el [29/09/2011]  
[http://sourceforge.net/project/shownotes.php?release\\_id=595815](http://sourceforge.net/project/shownotes.php?release_id=595815)
- [19]. PACKAGE: perfctr-2.6.35-1.el5.jp.kp.i386.rpm Consultado el [30/09/2011]  
<http://parasol.tamu.edu/people/jkp2866/myRPMS/PAPI/perfctr/old/perfctr-2.6.35-1.el5.jp.kp.i386.rpm.txt>
- [20]. OpenMP.org >> About the OpenMP ARB and OpenMP.org Consultado el [02/10/2011]  
<http://openmp.org/wp/about-openmp/>
- [21]. OpenMP.org >> OpenMP Compilers Consultado el [02/10/2011]  
<http://openmp.org/wp/openmp-compilers/>
- [22]. OpenMP 3.0-SummarySpec Consultado el [02/10/2011]  
<http://www.openmp.org/mp-documents/OpenMP3.0-SummarySpec.pdf>

- [23] PAPI Consultado el [02/10/2011]  
<http://icl.cs.utk.edu/papi/>
- [24] PAPI 3.5 USER'S GUIDE Consultado el [03/10/2011]  
[http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI\\_USER\\_GUIDE.htm](http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE.htm)
- [25] PAPI Forum Consultado el [03/10/2011]  
<http://icl.cs.utk.edu/papi/forum/>
- [26] Valgrind Home Consultado el [25/10/2011]  
<http://valgrind.org/>
- [27] Hardware Prefetching  
[http://suif.stanford.edu/papers/mowry92/subsection3\\_5\\_2.html](http://suif.stanford.edu/papers/mowry92/subsection3_5_2.html)
- [28] Optimizing Embedded System Performance — Impact of Data Prefetching on a Medical Imaging Application Consultado el [03/10/2011]  
[ftp://download.intel.com/technology/advanced\\_comm/315256.pdf](ftp://download.intel.com/technology/advanced_comm/315256.pdf)
- [29] AMIDE: a Medical Image Data Analysis Tool  
<http://amide.sourceforge.net/>
- [30] Debian – Details of package msr-tools Consultado el [04/10/2011]  
<http://packages.debian.org/unstable/admin/msr-tools>
- [31] Optimizing Application Performance on Intel Registro 0x1A0 Consultado el [05/10/2011]  
<http://software.intel.com/en-us/articles/optimizing-application-performance-on-intel-core-microarchitecture-using-hardware-implemented-prefetchers>
- [32] AMD Support página 434 Registro MSRC001\_1022 Consultado el [05/10/2011]  
[http://support.amd.com/us/Processor\\_TechDocs/31116.pdf](http://support.amd.com/us/Processor_TechDocs/31116.pdf)
- [33] GCC – the GNU Compiler Collection – GNU Project – Free Software Foundation (FSF) Consultado el [06/10/2011]  
<http://gcc.gnu.org/>
- [34] Optimize Options – Using the GNU Compiler Collection (GCC) Consultado el [06/10/2011]  
<http://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/Optimize-Options.html>
- [35] Predicting Cache Needs and Cache Sensitivity for Applications in Cloud Computing on CMP Servers with Configurable Caches.  
 Jacob Machina. Departamento de Informática. Universidad de Windsor, Windsor, Canadá.
- [36] Introduction to Parallel Architectures - Universidad de Texas Arlington Consultado el [11/11/2011]  
[www.cs.utah.edu/~mhall/cs4961f10/CS4961-L3.ppt](http://www.cs.utah.edu/~mhall/cs4961f10/CS4961-L3.ppt)
- [37] Computer Architecture: A Quantitative Approach Pg 362  
 John L. Hennessy, David A. Patterson
- [38] The Rutherford-Boeing Sparse Matrix Collection  
 RAL-TR-97-031  
 Ian S. Duff, Roger G. Grimes y John G. Lewis.
- [39] Harwell-Boeing Exchange Format Consultado el [08/10/2011]  
<http://math.nist.gov/MatrixMarket/formats.html#hb>
- [40] Matrix Market Exchange Formats Consultado el [08/10/2011]  
<http://math.nist.gov/MatrixMarket/formats.html#MMformat>
- [41] Matlab (ASCII) sparse matrix format Consultado el [09/10/2011]  
<http://bebop.cs.berkeley.edu/smc/formats/matlab.html>
- [42] BeBop Sparse Matrix Converter Consultado el [09/10/2011]  
<http://bebop.cs.berkeley.edu/smc/index.html#whatisit>
- [43] Direct Methods for Sparse Linear Systems, and the CSparse package Consultado el [09/10/2011]  
<http://www.cise.ufl.edu/research/sparse/CSparse/>
- [44] Direct Methods for Sparse Linear Systems. Timothy A. Davis  
 ISBN – 0898716136, Editor Society for Industrial and Applied Mathematics
- [45] Tim Davis: University of Florida Sparse Matrix Collection Consultado el [10/10/2011]  
<http://www.cise.ufl.edu/research/sparse/matrices/>
- [46] Simulated Annealing – GNU Scientific Library – Reference Manual Consultado el [10/10/2011]



[http://www.gnu.org/software/gsl/manual/html\\_node/Simulated-Annealing.html](http://www.gnu.org/software/gsl/manual/html_node/Simulated-Annealing.html)

[47] Clever Algorithms: Nature-Inspired Programming Recipes By Jason Bronlee Consultado el [10/10/2011]

<http://books.google.com/books?id=SESWXQphCUkC&lpg=PA169&dq=simulated%20annealing&pg=PA169#v=onepage&q=simulated%20annealing&f=false>

[48] revista enLinea – Optimización con recocido simulado para el problema de conjunto independiente Consultado el [12/10/2011]

<http://www.azc.uam.mx/publicaciones/enlinea2/3-2rec.htm>

[49] GNU Scientific Library – Reference – GNU Project – Free Software Foundation (FSF) Consultado el [13/10/2011]

<http://www.gnu.org/software/gsl/manual/index.html#dir>

[50] Simulated Annealing - GNU Scientific Library – Reference Manual Consultado el [15/10/2011]

[http://www.gnu.org/software/gsl/manual/html\\_node/Simulated-Annealing.html](http://www.gnu.org/software/gsl/manual/html_node/Simulated-Annealing.html)

[51] My program runs fine when measuring 1 or 2 events, but when I add more I get a -8 PAPI\_ECNFLCT error code. The error text says, "Event exists but cannot be counted due to hardware resource limitations". What does this mean? Consultado el [15/10/2011]

<http://icl.cs.utk.edu/papi/faq/index.html#170> Consultado el [14/11/2011]

[52] Getting Ready to Meet Core2Duo : Core architecture unleashed

[http://www.xbitlabs.com/articles/cpu/display/core2duo-preview\\_9.html](http://www.xbitlabs.com/articles/cpu/display/core2duo-preview_9.html)

[53] Matrix: Oberwolfach/t3dh\_a Consultado el [16/10/2011]

[http://www.cise.ufl.edu/research/sparse/matrices/Oberwolfach/t3dh\\_a.html](http://www.cise.ufl.edu/research/sparse/matrices/Oberwolfach/t3dh_a.html)

[54] Matrix: Norris/stomach Consultado el [16/10/2011]

<http://www.cise.ufl.edu/research/sparse/matrices/Norris/stomach.html>

[55] Matrix: Boeing/msc10848 Consultado el [16/10/2011]

<http://www.cise.ufl.edu/research/sparse/matrices/Boeing/msc10848.html>

[56] Matrix: Simon/appu Consultado el [16/10/2011]

<http://www.cise.ufl.edu/research/sparse/matrices/Simon/appu.html>

[57] Matrix: Norris/heart1 Consultado el [16/10/2011]

<http://www.cise.ufl.edu/research/sparse/matrices/Norris/heart1.html>

[58] Matrix: FEMLAB/ns3Da Consultado el [16/10/2011]

<http://www.cise.ufl.edu/research/sparse/matrices/FEMLAB/ns3Da.html>

[59] TagCrow: make your own tag cloud from any text Consultado el [27/10/2011]

<http://tagcrowd.com/>

[60] Cesga – Hours of calculation Consultado el [01/11/2011]

<https://www.cesga.es/en/servicios/computacion/horas-calculo#Tarifas>

[61] Order SPEC Benchmarks

<http://www.spec.org/order.html>

[62] SPEC 2006 Consultado el [12/11/2011]

<http://www.spec.org/cpu2006/>

[63] Threads – PAPI Docs Consultado el [02/11/2011]

<http://icl.cs.utk.edu/projects/papi/wiki/Threads>

[64] Papers – 2<sup>nd</sup> USENIX WINDOWS NT Symposium Consultado el [02/11/2011]

[http://www.usenix.org/publications/library/proceedings/usenix-nt98/full\\_papers/zabatta/zabatta.html/zabatta.html](http://www.usenix.org/publications/library/proceedings/usenix-nt98/full_papers/zabatta/zabatta.html/zabatta.html)

[65] PAPI – Perfometer doc and GUI Consultado el [24/10/2011]

<http://icl.cs.utk.edu/papi/custom/index.html?lid=51&slid=70>

## Apéndice A: PAPI Guía de instalación y uso

---

En las siguientes páginas se muestra un manual sobre la instalación y el uso de la librería PAPI. Fue realizado en las etapas iniciales del proyecto como trabajo dirigido y en él se abordan de forma más específica los problemas que pueden derivar de instalar y/o utilizar las librerías PAPI.

Deseamos que en ellos encuentre la base necesaria para arrancar con el uso de la herramienta. Junto con el código del proyecto también se adjuntará el código utilizado como ejemplo en este manual.

PAPI

# PERFORMANCE APPLICATION PROGRAMMING INTERFACE:

GUÍA DE INSTALACIÓN Y USO



Universidad Carlos III Madrid  
Trabajo Dirigido

Alumno: Antonio Díaz Ponce

Ingeniería Informática

## Índice

1.- Introducción.....	133
2.- Instalación:.....	134
2.1 Recompilar el kernel .....	136
2.1.1 Descargar y descomprimir fuentes.....	136
2.1.2 Descargar y parchear el kernel con “PerfCtr”.....	136
3.- Uso.....	141
3.1 Eventos.....	141
3.1.1 Eventos nativos .....	141
3.1.1 Eventos preestablecidos (preset events).....	143
3.2 Interfaz de contadores PAPI .....	145
3.2.1 Interfaz de alto nivel .....	145
3.2.2 Interfaz de bajo nivel .....	148
3.3 Temporizadores PAPI.....	151
3.4 Funcionalidades avanzadas.....	153
3.5 Errores.....	155
4.- example.c.....	156
Apéndice .....	157

## Índice de códigos y capturas

Código 1: Salida del comando papi_native_avail.....	141
Código 2: Código en el que se añade un evento nativo .....	142
Código 3: Salida del comando papi_avail .....	143
Código 4: Código en el que se utiliza un evento preestablecido .....	144
Código 5: Ejemplo interfaz alto nivel parte I.....	145
Código 6: Ejemplo interfaz alto nivel parte II .....	146
Código 7: Ejemplo interfaz alto nivel parte III.....	146
Código 8: Salida de ejecución de la interfaz de alto nivel.....	147
Código 9: Ejemplo interfaz bajo nivel parte I.....	148
Código 10: Ejemplo interfaz bajo nivel parte II .....	149
Código 11: Ejemplo interfaz bajo nivel parte III.....	149
Código 12: Salida de ejecución de la interfaz de bajo nivel .....	150
Código 13: Ejemplo de utilización de temporizadores virtuales .....	151
Código 14: Salida de un programa en el que usamos más contadores de los existentes.....	155
Código 15: Salida del script.sh .....	156

## 1.- Introducción

En este manual se tratarán los pasos a seguir a la hora de instalar las librerías de PAPI, así como algunos de los problemas con los que el usuario se puede encontrar durante la fase de instalación o de uso de estas librerías.

Como su propio nombre indica PAPI, es un API de programación para la lectura de contadores hardware. Su uso es realmente sencillo, aunque los primeros pasos pueden ser en algunos momentos desesperantes, esto se debe a los problemas que genera según la arquitectura y la plataforma sobre la que se use.

Para cualquier duda, que no quede resuelta en este manual, es recomendable utilizar el foro de PAPI <http://icl.cs.utk.edu/papi/forum/index.php>, en el que los usuarios contribuyen entre sí resolviendo dudas.

PAPI posee también varios manuales que actualizan a la vez que sacan nuevas versiones. Gran parte del apartado “3 USO” está basando en éstos. A continuación, un par de enlaces a los manuales:

- [http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI\\_USER\\_GUIDE\\_23.htm](http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE_23.htm)
- [http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI\\_USER\\_GUIDE\\_306.htm](http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI_USER_GUIDE_306.htm)

En ocasiones también es aconsejable leer la FAQ (preguntas más frecuentes):

- <http://icl.cs.utk.edu/papi/faq/index.html>

### 1.1 Arquitectura hardware y software

Las pruebas y ejecuciones que se muestran en el siguiente manual se realizaron con un ordenador portátil **TOSHIBA Satellite A200** – 1dy. Cuyas características principales son (salida obtenida mediante el programa CPU-Z):

1 procesador Intel Mobile Core 2 Duo T7100 a 1.80GHz  
L1 Data cache → 2 x 32 KBytes, 8-way set associative, 64-byte line size  
L1 Instruction cache → 2 x 32 KBytes, 8-way set associative, 64-byte line size  
L2 cache → 2048 KBytes, 8-way set associative, 64-byte line size

#### Chipset

Northbridge → Intel GM965 rev. C0  
Southbridge → Intel 82801HBM (ICH8-ME) rev. B1  
Memory Type → DDR2  
Memory Size → 1024 Mbytes

En cuanto al sistema operativo utilizado será **Fedora 12**, las razones se explican en el siguiente apartado, Instalación. La interfaz elegida es indiferente, en este caso se utilizó Gnome.

## 2.- Instalación:

Para realizar la instalación seguiremos los pasos del archivo `INSTALL.txt` que se encuentra en la carpeta raíz de `papi`. Los pasos a seguir son, dentro de esta carpeta realizar las siguientes instrucciones:

- `./configure`
- `make`

El siguiente paso podremos seleccionar cualquiera de las siguientes opciones, probablemente requiera autenticarse como superusuario (`sudo`, o `su -`, según la plataforma en la que se ejecute):

- `make install` → para instalar las librerías y los archivos cabecera (.h).
- `make install-man` → para instalar el manual de PAPI.
- `make install-test` → para instalar los tests.
- `make install-all` → para instalar todo.

Desde este manual se recomienda encarecidamente, utilizar la versión **12 de Fedora**, o cualquier otra versión que no requiera la recompilación del kernel, ya que es una labor tediosa y complicada.

Para probar que todo ha funcionado correctamente podremos ir a la carpeta `/src/examples` y probar algunos de los ejemplos. Además en si en la carpeta `src`, hacemos directamente `make test`, ejecutaremos un simple test donde podremos verificar si la instalación se ha realizado de forma correcta y sin la necesidad de compilar.

A la hora de compilar debemos acordarnos de incluir el argumento `-lpapi`, para que compile enlazando con las librerías de PAPI. Puede que al compilar se produzca algún error, para solucionarlo añadiremos en el fichero `.bashrc`, la siguiente línea:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
```

De esta forma enlazaremos las librerías de PAPI, que después de la instalación se encuentran en `/usr/local/lib`.

Si tenemos problemas a la hora de realizar los pasos, en el archivo `INSTAL.txt`, tenemos una serie de consejos según cual sea el sistema operativo y el CPU que estemos usando.

El kernel de otras versiones de Linux, como por ejemplo Ubuntu 9.04, requiere estar parcheado y recompilado con el parche PerfCtr. Al utilizar una de estas versiones no parcheadas recibiremos un:

Papi configure error – performance monitoring interface

En caso de no poseer el kernel parcheado, la instalación del parche viene descrita en `papi/src/perfctr-.2.x.x`, el archivo que se utilizará para ello es `update-kernel`. Además en la sección 2.1 veremos como se realiza la recompilación del kernel paso a paso.

Para conocer si hemos realizado la instalación finalmente de forma correcta podemos ejecutar el comando `make test`, dentro de la carpeta `src` de PAPI, que realizará un pequeño test y obtendremos distintos valores.

## 2.1 Recompilar el kernel

Si deseamos mantener el sistema operativo, y decidimos recompilar el kernel deberemos seguir una rigurosa secuencia de pasos, por la que, probablemente, nos iremos encontrando más problemas, a la hora de recompilar, etc. Por lo que de nuevo se desaconseja seguir este camino.

Gran parte de la información correspondiente a este apartado fue obtenida de la web: [La plaga Tux: Instalación de librerías PAPI en linux](http://plagatux.es/2008/01/instalacion-de-librerias-papi-en-linux/)

(<http://plagatux.es/2008/01/instalacion-de-librerias-papi-en-linux/>)

También recomendamos echar un vistazo al siguiente tutorial, con el que el profesor Francisco Javier García Blas consiguió actualizar el kernel en un AMD Opteron:

<http://www.cse.ohio-state.edu/~hartonoa/papi-ubuntu.txt>

### 2.1.1 Descargar y descomprimir fuentes

Para descargar las fuentes de las librerías tan solo tenemos que acceder a la página principal del proyecto [PAPI](#), acceder a la zona “software” y descargar la última versión (3.9.0). Os dejo también el enlace directo a la descarga para más facilidad ([aquí](#) → <http://icl.cs.utk.edu/projects/papi/downloads/papi-c-3.9.0.tar.gz>).

Descomprimos el directorio donde nos plazca (por ejemplo en vuestro \$HOME).

```
~$ tar -xvf papi-c-3.9.0.tar.gz
```

### 2.1.2 Descargar y parchear el kernel con “PerfCtr”

En la documentación existente dentro de la carpeta que acabamos de descomprimir, se nos señala los requisitos necesarios para cada tipo de procesador y/o arquitectura de pc (tanto software como hardware). A continuación os indicaré como realizar todos los pasos necesarios para cualquier distribución linux/x86 con procesadores Intel y AMD.

Para las plataformas linux/x86 se requiere que el kernel del sistema esté parcheado y recompilado con el parche **PerfCtr**. Los parches requeridos y las instrucciones de instalación se proporcionan en el directorio `papi/src/perfctr-x.y` (donde x e y indican la versión del kernel). El parche más reciente que existe dentro del paquete es para la versión 2.6.18 del kernel de linux, por lo que tendremos que bajarnos dicha versión de la web <http://www.kernel.org>, donde se encuentran las fuentes “vanilla” (sin ningún tipo de parche aplicado) de todas las versiones de los kernels de linux. Podemos



descargar dichas fuentes de cualquier mirror, aquí un enlace para descargar directamente:  
<http://www.eu.kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.tar.bz2>.

Ahora realizamos los siguientes pasos que se enumeran:

1. Descomprimir las fuentes del kernel

```
~$ tar -xvfj linux-2.6.18.tar.bz2
```

2. Ahora pasamos a copiar el directorio recién creado a la ruta /usr/src y creamos un enlace simbólico tal y como se muestra (En el primer paso nos autenticamos como superusuario, para no tener que poner sudo en todos los comandos siguientes que introduzcamos):

```
~$ sudo su
~$ cp linux-2.6.18 /usr/src/ -R
~$ cd /usr/src
~$ ln -s linux-2.6.18/ linux
```

#Tener cuidado si ya tenemos este enlace simbólico creado ya que cuando compilamos programas, a veces se hace referencia a esta ruta.

3. Instalamos algunos paquetes que nos harán falta para la recompilación del kernel

```
~$ apt-get install build-essential kernel-package libncurses5-dev yaird
```

4. Accedemos al directorio donde se encuentra el kernel que hemos descomprimido, realizamos **make mrproper** para eliminar todos los archivos objeto y posibles dependencias del árbol de fuentes del núcleo.

```
~$ cd /usr/src/linux
~$ make mrproper
```

5. Aplicamos el parche PerfCtr que se encuentra dentro del directorio apropiado generado tras haber descomprimido el archivo que contiene la librería PAPI. Para ello lo hacemos con el script que viene dentro de dicha carpeta:

```
~$ /ruta-usuario/papi-c-3.9.0/src/perfctr-2.6.x/update-kernel
```

6. Puede ser que en este paso nos de un error por una discordancia entre el tipo de Shell especificado en el script y el tipo de shell que usamos (generalmente **bash**). En caso de que no se haya aplicado correctamente el parche, debemos editar dicho script y modificar la primera línea del mismo por:

```
#!/bin/bash
```

7. Una vez que se aplica correctamente el parche pasamos a la configuración de los módulos del kernel. Esto se puede realizar de varios modos (make config, make menuconfig, make xconfig o make gconfig). Si vamos a utilizar la distribución Ubuntu es recomendable hacerlo con **make gconfig**. Los módulos que debemos marcar obligatoriamente de forma estática (con YES) son:

- Procesor type and features – Performance-monitoring counters support (PERFCTR)
  - Init-time hardware test (PERFCTR\_INIT\_TEST)
  - Virtual performance counters support (PERFCTR\_VIRTUAL)
  - Global performance counters support (PERFCTR\_GLOBAL)
- File Systems – Ext3 Journaling file system support (EXT3\_FS) (O el sistema de archivos que utilicemos)
- Device Drivers – SCSI device support – SCSI low-level drivers – Serial ATA (SATA) support (Marcar los módulos SATA que nos interesen: según la marca de placa base).
- La activación o no del resto de módulos queda fuera del objetivo de esta entrada.

8. Para construir el paquete .deb que nos servirá posteriormente para instalar nuestro kernel, usaremos la función make-kpkg. Esta orden sustituye a las clásicas ordenes: **make dep**, **make clean**, **make bzImage** y **make modules**:

```
~$ make-kpkg clean
```

```
~$ make-kpkg --append-to-version=.XXXX --initrd kernel_image
```

Donde .XXXX indica solamente una adición (como puede ser la fecha actual) que aparecerá en el nombre del paquete y del kernel.

Llegados a este paso, con la versión del kernel que se indica puede llegar aparecer el siguiente error:

```
scripts/mod/sumversion.c:384: error: 'PATH_MAX' no se declaró aquí (primer uso en esta función)
```

```
scripts/mod/sumversion.c:384: error: (Cada identificador no declarado solamente se reporta una vez  
scripts/mod/sumversion.c:384: error: para cada funcion en la que aparece.)
```

Para solucionarlo deberemos incluir `#include` en dicho archivo `sumversion.c`.

Si después de este fallo obtenemos un error relacionado con `#elif`, lo que deberemos hacer es sustituir el último `#elif` por `#else` en el archivo `drivers/video/sstfb.c`

9. Una vez generada la imagen (se genera en `/usr/src/`) tan solo tenemos que instalarla con el siguiente comando:

```
~$ dpkg -i nombre-paquete.deb
```

Automáticamente se copian todos los archivos necesarios en `/boot/` y se actualiza el grub para poder seleccionar en el próximo arranque de nuestro equipo la nueva imagen de kernel recién instalada.

10. Arrancamos el sistema con el nuevo kernel.

Llegados a este punto personalmente tuve un problema al iniciar el sistema operativo nuevo, y dejé de seguir intentándolo por este camino, puesto que comprobé que con Fedora 12 no era necesario recompilar el kernel. El error decía lo siguiente.

```
error getting signalfd  
udevd[1036]: error getting signal fd
```

Probablemente se trata de algún tipo de incompatibilidad entre el ordenador y el kernel descargado, pero es una simple conjetura.

11. El kernel-side del parche PerfCtr está implementado como un driver de dispositivo de caracteres, al que se le ha asignado los números 10 y 182. La primera vez que se instala el paquete, un fichero especial representando a dicho dispositivo debe ser creado. Como root, ejecutamos:

```
~$ mknod /dev/perfctr c 10 182  
  
~$ chmod 644 /dev/perfctr  
  
~$ chown miusuario:miusuario /dev/perfctr
```

Si en futuros reinicio observamos que no podemos utilizar correctamente la librería PAPI, puede que sea porque se han perdido los permisos de dicho dispositivo y tengamos que realizar de nuevo la segunda instrucción indicada y tercera instrucción indicada (donde `miusuario`, es el usuario con el que vais a trabajar).

12. Para instalar la librería realizamos dentro del directorio de perfctr (/papi/src/perfctr-2.6.x):

```
~$ make PREFIX=/usr/ install
```

## 3.- Uso

A continuación, en este tercer apartado trataremos los distintos componentes que poseen este API, para el funcionamiento de los contadores hardware.

### 3.1 Eventos

En este conjunto de código que forma PAPI hay un elemento básico y fundamental que son los eventos. Los eventos son ocurrencias o señales específicas relativas a una ejecución. Los contadores hardware existen como un pequeño conjunto de registros que cuentan este tipo de eventos, como por ejemplo: los fallos cache, operaciones en punto flotante; todo esto se realiza mientras el procesador ejecuta el programa deseado.

#### 3.1.1 Eventos nativos

Los eventos nativos comprenden el conjunto de todos los eventos que pueden ser contados por esa CPU. Generalmente el número de eventos nativos suele ser mayor que el número de eventos preestablecidos por PAPI (preset events).

Para conocer los eventos nativos que poseemos en la máquina en la que deseamos utilizar esta API, podremos ejecutar el comando “`papi_native_avail`”. Obtendremos una salida como la siguiente:

```
Available native events and hardware information.
```

```
-----
PAPI Version           : 4.0.0.2
```

```
 #(Características del ordenador)
-----
```

```
The following correspond to fields in the PAPI_event_info_t structure.
```

```
Event Code   Symbol   | Long Description |
```

```
.....
```

```
0x4000000f  DTLB_MISSES | Memory accesses that missed the DTLB |
4000040f    :ANY   | Any memory access that missed the DTLB |
4000080f    :MISS_LD | DTLB misses due to load operations |
4000100f    :LO_MISS_LD | L0 DTLB misses due to load operations |
4000200f    :MISS_ST | DTLB misses due to store operations |
-----
```

```
0x40000010  MEMORY_DISAMBIGUATION | Memory disambiguation |
40000410    :RESET  | Memory disambiguation reset cycles |
40000810    :SUCCESS | Number of loads that were successfully |
-----
```

Código 1: Salida del comando `papi_native_avail`

Para utilizar eventos nativos de forma efectiva deberemos estar familiarizados con la plataforma que estamos utilizando. PAPI provee acceso a los eventos nativos en todas las plataformas posibles mediante la interfaz de bajo nivel (low-level interface).

Los eventos nativos utilizan la misma interfaz que se usa para configurar los eventos preestablecidos, pero en ocasiones si la definición de los eventos nativos no ha sido realizada, este evento nativo deberá utilizar un código de eventos para poder ser utilizado, este código de evento es el que aparece en la columna de la izquierda en la salida del comando `papi_native_avail`.

**Los códigos de los eventos nativos y sus respectivos nombres son dependientes de la plataforma**, por lo que es muy probable que un programa realizado con ciertos códigos en una plataforma no funcione en otra.

Los eventos nativos se especifican como argumentos para la funcion de bajo nivel `PAPI_add_event`, de manera similar a como se hace con los eventos preestablecidos. En el siguiente ejemplo de código, vemos como se obtiene el código de un evento nativo y es añadido a un conjunto de eventos (`eventSet`) que veremos más adelante.

```
#include <papi.h>
#include<stdio.h>
main()
{
    int retval, EventSet = PAPI_NULL;
    unsigned int native = 0x0;
    PAPI_event_info_t info;
    /* Initialize the library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
        printf("PAPI library init error!\n");
        exit(1);
    }
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        handle_error(1);

    /* Find the first available native event */
    native = NATIVE_MASK | 0;
    if (PAPI_get_event_info(native, &info) != PAPI_OK) {
        if (PAPI_enum_event(&native, 0) != PAPI_OK)
            handle_error(1);
    }
    /* Add it to the eventset */
    if (PAPI_add_event(EventSet, native) != PAPI_OK)
        handle_error(1);
}
```

Código 2: Código en el que se añade un evento nativo

En el código final entregado, “example.c” contiene la utilización de un evento nativo y un evento preestablecido a la vez.

### 3.1.1 Eventos preestablecidos (preset events)

Los eventos preestablecidos o también conocidos como los eventos predefinidos, son el conjunto de los eventos más relevantes a la hora de realizar aplicaciones que miden el rendimiento del ordenador. Estos eventos se encuentran normalmente en muchas CPUs que proveen de contadores y dan acceso a históricos de memoria, eventos sobre protocolos de coherencia caché, ciclos e instrucciones...

Además, a todos estos eventos se les ha asignado previamente un nombre simbólico. Por ejemplo, el total de ciclos sería PAPI\_TOT\_CYC.

Las librerías de PAPI poseen alrededor de 100 eventos preestablecidos. Éstos se encuentran definidos en el archivo de cabecera, `papiStdEventDefs.h`. Al igual que sucedía con los eventos nativos, podemos conocer qué eventos preestablecidos soporta nuestro sistema, pero utilizando el comando “`papi_avail`”. En el siguiente cuadro de texto vemos una posible salida de este comando.

```
Available events and hardware information.
-----
PAPI Version           : 4.0.0.2
#(Características del ordenador)
-----
The following correspond to fields in the PAPI_event_info_t structure.

    Name           Code      Avail  Deriv  Description (Note)
PAPI_L1_DCM       0x80000000   Yes    No    Level 1 data cache misses
PAPI_L1_ICM       0x80000001   Yes    No    Level 1 instruction cache misses
PAPI_L2_DCM       0x80000002   Yes    Yes   Level 2 data cache misses
PAPI_L2_ICM       0x80000003   Yes    No    Level 2 instruction cache misses
PAPI_L3_DCM       0x80000004   No     No    Level 3 data cache misses
PAPI_L3_ICM       0x80000005   No     No    Level 3 instruction cache misses
PAPI_L1_TCM       0x80000006   Yes    No    Level 1 cache misses
PAPI_L2_TCM       0x80000007   Yes    No    Level 2 cache misses
PAPI_L3_TCM       0x80000008   No     No    Level 3 cache misses
```

Código 3: Salida del comando `papi_avail`

Como se puede observar en el cuadro de texto, el comando `papi_avail` nos ofrece toda la lista de los eventos preestablecidos, con su nombre, su código, nos dice además si está disponible o no, si se trata de un evento derivado y su descripción.

La semántica exacta de los eventos es dependiente de la plataforma. Los nombres preestablecidos se mapean en torno a variables de tal forma que se intenta mapear el mayor número posible de eventos en las distintas plataformas. Debido a las diferencias de implementación en el hardware, no es viable la comparación directa entre los resultados obtenidos en distintas plataformas, mediante eventos preestablecidos.

### Eventos derivados

Los contadores hardware cuentan eventos de bajo nivel que pueden ser directamente medidos mediante hardware. Frecuentemente estos eventos de bajo nivel deben ser combinados para conseguir eventos preestablecidos de PAPI. Esta combinación lineal de eventos de bajo nivel se denomina eventos derivados PAPI. Se suelen formar mediante la resta de dos eventos nativos, pero en ocasiones los eventos derivados pueden formarse a partir de 4 o más términos.

La utilización de eventos derivados puede dar problemas a la hora de utilizar lo que se conoce como multiplexación, que se verá más adelante en este manual.

En la siguiente figura veremos un ejemplo de aplicación de eventos preestablecidos, en concreto se utilizará el evento PAPI\_TOT\_INS, que nos indica el total de instrucciones que se han ejecutado.

```
#include <papi.h>
#include <stdio.h>
main() {
    int EventSet = PAPI_NULL;
    unsigned int native = 0x0;
    int retval, i;
    PAPI_preset_info_t info;
    PAPI_preset_info_t *infostructs;
    /* Initialize the library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
        fprintf(stderr, "PAPI library init error!\n"); exit(1);
    }
    /* Check to see if the preset, PAPI_TOT_INS, exists */
    if (PAPI_query_event (PAPI_TOT_INS) != PAPI_OK) {
        fprintf (stderr, "No instruction counter? How lame.\n"); exit(1);
    }
    /* Get details about the preset, PAPI_TOT_INS */
    if (PAPI_get_event_info(PAPI_TOT_INS, &info) != PAPI_OK) {
        fprintf (stderr, "No instruction counter? How lame.\n");
        exit(1);
    }
    if (info.count > 0)
        printf ("This event is available on this hardware.\n");
    if (info.flags & PAPI_DERIVED)
        printf ("This event is a derived event on this hardware.\n");
    /* Count the number of available preset events between
       PAPI_TOT_INS and the end of the preset list */
    retval = 0;
    i = PAPI_TOT_INS;
    while (PAPI_enum_event(&i, TRUE) == PAPI_OK) {
        retval++;
    }
}
```

Código 4: Código en el que se utiliza un evento preestablecido



## 3.2 Interfaz de contadores PAPI

### 3.2.1 Interfaz de alto nivel

La interfaz de alto nivel provee la posibilidad de arrancar, parar y leer los contadores de una lista específica de eventos. Esto significa que los programadores podrán tomar medidas utilizando únicamente eventos preestablecidos.

Algunos de los beneficios de utilizar la interfaz de alto nivel, en comparación con la de bajo nivel, es que es más fácil de utilizar y requiere menos configuración (llamadas adicionales). La interfaz de alto nivel también puede ser utilizada junto con la de bajo nivel, de hecho, la interfaz de alto nivel realiza llamadas a la de bajo nivel. Sin embargo, la interfaz de alto nivel es capaz, por sí misma, de acceder a los eventos contables simultáneamente durante la ejecución.

Existen ocho funciones que representan la interfaz de alto nivel, y que permiten al usuario acceder a eventos hardware. Se puede acceder a las variables tanto en C como en Fortran, en este manual solo se tratarán las llamadas en C, para ver más información utilice las referencias del apartado 1.- Introducción.

Para ver exactamente el funcionamiento de este tipo de interfaz vamos a ver un ejemplo detallado de cómo se toman varias lecturas de varios contadores, el código completo y sin comentarios se puede tomar del apartado Apéndice.

El primer paso será incluir las librerías que vayamos a necesitar, entre ellas `papi.h`, en concreto en esta prueba se tomarán medidas sobre:

`PAPI_TOT_INS` → Número total de instrucciones.

`PAPI_FP_INS` → Instrucciones con operaciones en punto flotante.

`PAPI_TOT_CYC` → Número total de ciclos.

```
#include <papi.h>
#include <stdio.h>

#define NUM_EVENTS 3

main()
{
    int i = 0, j = 0;

    int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_FP_INS, PAPI_TOT_CYC};

    long_long values[NUM_EVENTS];

    char EventCodeStr[PAPI_MAX_STR_LEN];
```

Código 5: Ejemplo interfaz alto nivel parte I

El siguiente paso será comenzar a leer de los contadores que tenemos en la variable `Events`, para ello utilizaremos `PAPI_start_counters()`. Esta función inicializa las librerías de PAPI en caso de ser necesario. En la interfaz de bajo nivel la función `start` no inicializa las librerías.

```
/* Start counting events */
if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK) {
    printf("1-----error\n");
}
```

Código 6: Ejemplo interfaz alto nivel parte II

El siguiente paso será realizar una lectura previa de los contadores, lo haremos mediante la función `PAPI_read_counters`, y mostramos los contadores con el bucle final.

Con la función `PAPI_event_code_to_name` podemos conocer el nombre del contador que se encuentra en el array `Events[i]`, y así imprimirlo.

```
/* Read the counters */
if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK) {
    printf("2-----error\n");
}

/*Print results*/
printf("-->1 (Read, starting)\n");
for(i=0; i<NUM_EVENTS; i++){
    PAPI_event_code_to_name(Events[i], EventCodeStr);
    printf("%s = %lld\n", EventCodeStr, values[i]);
}
```

Código 7: Ejemplo interfaz alto nivel parte III

El siguiente paso será realizar la ejecución del programa, después se introducen las mismas líneas de código del cuadro anterior y veremos como ahora los valores de los contadores son mucho más altos

El código completo se encuentra en la sección Apéndice (CODIGO DE INTERFAZ DE ALTO NIVEL).

Al finalizar es conveniente utilizar la función `PAPI_stop_counters`, que como su propio nombre indica, para de leer de los contadores.

La salida obtenida fue la siguiente:

```
-->1 (Read, starting)
PAPI_TOT_INS = 2857
PAPI_FP_INS = 4
PAPI_TOT_CYC = 9081
-->2 (Stop, end)
PAPI_TOT_INS = 7407427252
PAPI_FP_INS = 19
PAPI_TOT_CYC = 12055252844
```

Código 8: Salida de ejecución de la interfaz de alto nivel

La ejecución ha requerido bastantes ciclos a la CPU, en comparación con los ciclos al inicio del programa, cuando simplemente estábamos inicializando valores.

### 3.2.2 Interfaz de bajo nivel

La interfaz de bajo nivel gestiona los eventos hardware en lo que se define como conjuntos de eventos (Event Sets). Es utilizado principalmente por programadores más expertos que pretenden tener un mayor control sobre la interfaz PAPI. Al contrario que la interfaz de alto nivel, **sí permite la utilización simultánea de eventos nativos y eventos preestablecidos**. Otras ventajas de la utilización de la interfaz de bajo nivel, pueden ser la obtención de información del ejecutable, así como utilizar las opciones de multiplexación (ver apartado 3.4 multiplexación) y manejo de desbordamiento.

La interfaz de bajo nivel puede ser utilizada junto con la de alto nivel, siempre asegurándonos de que las librerías de PAPI han sido inicializadas (en la interfaz de alto nivel se realiza de forma implícita con el método start).

Al igual que hicimos con la interfaz de alto nivel, veremos un ejemplo completo sobre la lectura de un conjunto de contadores, pero esta vez basado en la interfaz de bajo nivel. El código completo sin comentarios lo podremos encontrar en el apartado final del documento “Apéndice” (CODIGO DE INTERFAZ DE BAJO NIVEL).

El primer paso es, por tanto, inicializar las librerías y crear el conjunto de eventos o Event Set.

```
int retval, EventSet=PAPI_NULL;

char EventCodeStr[PAPI_MAX_STR_LEN];

/* Initialize the PAPI library */
retval = PAPI_library_init(PAPI_VER_CURRENT);
if (retval != PAPI_VER_CURRENT) {
    printf("Error(%d): PAPI library init error!\n", retval);
}

/* Create the Event Set */
retval = PAPI_create_eventset(&EventSet);
if (retval != PAPI_OK) {
    printf("Error: PAPI_create_eventset\n");
}
```

Código 9: Ejemplo interfaz bajo nivel parte I

Una vez creado el conjunto de eventos, el siguiente paso será ir añadiendo uno a uno todos los eventos que deseemos probar, en este ejemplo se van a añadir dos eventos preestablecidos:

PAPI\_L2\_LDM que indica los fallos que se han producido en operaciones de carga.

PAPI\_L2\_STM que indica el número de fallos que se han producido en operaciones de guardado.

```
retval = PAPI_add_event(EventSet, PAPI_L2_LDM);
if (retval != PAPI_OK){
    printf("Error (%d): PAPI_add_event 1\n", retval);
}

retval = PAPI_add_event(EventSet, PAPI_L2_STM);
if (retval != PAPI_OK){
    printf("Error (%d): PAPI_add_event 2\n", retval);
}

/* Start counting events in the Event Set */
if (PAPI_start(EventSet) != PAPI_OK){
}
```

Código 10: Ejemplo interfaz bajo nivel parte II

En este caso si quisiésemos añadir un evento nativo, por ejemplo, L2\_LINES\_IN, tendríamos que ver en la salida del comando `papi_native_avail`, el código de este evento. Para el caso del ordenador donde se ejecutaron las pruebas el código era 0x4000001b, simplemente deberíamos modificar PAPI\_L2\_LDM por este número en hexadecimal y ya estaríamos tomando medidas del número de fallos que se producen en el nivel 2 de la memoria caché.

Al final del anterior código comenzamos a tomar medidas con `PAPI_start`, salvando las distancias, a partir de este punto el código entre la interfaz de alto nivel y de bajo nivel es bastante similar.

```
/* Read the counting events in the Event Set */
if (PAPI_read(EventSet, values) != PAPI_OK){
}

printf("-->After reading the counters\n");
for(i=0; i<NUM_EVENTS; i++){
    printf("valor %d = %lld\n", i, values[i]);
}

/* Reset the counting events in the Event Set */
if (PAPI_reset(EventSet) != PAPI_OK){
}
```

Código 11: Ejemplo interfaz bajo nivel parte III

Simplemente se realiza una lectura y se muestra el valor del array de `long_long`, `values`, la siguiente sentencia realiza un reset sobre los contadores para ponerlos a cero.

En el resto de código se utilizan más funciones pero las que hemos visto aquí son básicamente las más vitales.

La salida que produce este segundo ejemplo es:

```
-->After reading the counters
valor 0 = 32
valor 1 = 7
-->After adding the counters
valor 0 = 37
valor 1 = 12
-->After stopping the counters
valor 0 = 14
valor 1 = 7
```

Código 12: Salida de ejecución de la interfaz de bajo nivel

Para el segundo caso se utilizó la función `PAPI_accum(EventSet, values)`, que añade a los valores anteriores los valores de la nueva lectura. Después se realiza un `PAPI_reset`, donde todos los eventos vuelvan a valer 0 y se vuelve a contar de nuevo, de ahí que la última lectura sea menor.

### 3.3 Temporizadores PAPI

Los medidores de tiempo PAPI utilizan los temporizadores más precisos disponibles en la plataforma. Estos temporizadores pueden utilizarse tanto para obtener tanto el tiempo real como el tiempo virtual. El tiempo real mide todo el tiempo que ha transcurrido (como un reloj de pared), y el tiempo virtual solo funciona cuando el procesador está ejecutando en modo usuario.

Para obtener una medida del tiempo real lo haremos mediante las funciones `PAPI_get_real_cyc()` y `PAPI_get_real_usec()`, que nos devolverán el tiempo real en ciclos y en microsegundos respectivamente. En caso de desear obtener el tiempo virtual utilizaremos las funciones `PAPI_get_virt_cyc()` y `PAPI_get_virt_usec()`.

A continuación, se presenta un pequeño extracto de código, obtenido del manual de PAPI.

```
#include <papi.h>
main()
{
    long_long start_cycles, end_cycles, start_usec, end_usec;
    int EventSet = PAPI_NULL;

    if (PAPI_library_init(PAPI_VER_CURRENT) != PAPI_VER_CURRENT)
        exit(1);
    /* Gets the starting time in clock cycles */
    start_cycles = PAPI_get_virt_cyc();
    /* Gets the starting time in microseconds */
    start_usec = PAPI_get_virt_usec();
    /* Create an EventSet */
    if (PAPI_create_eventset(&EventSet) != PAPI_OK)
        exit(1);

    /* Gets the ending time in clock cycles */
    end_cycles = PAPI_get_virt_cyc();
    /* Gets the ending time in microseconds */
    end_usec = PAPI_get_virt_usec();
    printf("Virtual clock cycles: %lld\n", end_cycles - start_cycles);
    printf("Virtual clock time in microseconds: %lld\n", end_usec - start_usec);
}
```

Código 13: Ejemplo de utilización de temporizadores virtuales

En el apartado ejemplo entregado `example.c` se hace utilización de estos temporizadores, la salida que vemos en el apartado 4, se corresponde a dicha utilización, en dicho caso los resultados se encuentran divididos entre 1000.

A la hora de tomar tiempos en `example.c` se realiza de la siguiente manera, se toman tres medidas de tiempo T1, T2 y T3. T1 y T2 se realizan antes y después de la ejecución respectivamente, y T3 se realiza de forma consecutiva a T2, como se muestra a continuación.

T1  
Ejecución  
T2  
T3

Finalmente para obtener el tiempo final de ejecución se utiliza la fórmula:

$$T_{\text{total}} = (T_2 - T_1) - (T_3 - T_2) = 2T_2 - T_1 - T_3$$



### 3.4 Funcionalidades avanzadas

El conjunto de librerías de PAPI es capaz de proveer varias funcionalidades bastante interesantes, entre las que se encuentran multiplexación, uso de contadores en programas paralelos (tanto en hilos como en MPI y OpenMP), control de desbordamiento y generación estadísticas mediante interrupciones.

En este manual sólo trataremos la funcionalidad de la multiplexación, en caso de querer consultar algunas de las opciones avanzadas existentes se recomienda utilizar las fuentes del apartado 1.- Introducción.

#### 3.4.1 Multiplexación

La multiplexación permite que se pueda monitorizar un mayor número de eventos de los que permite la plataforma en la que se ejecuta el programa. Cuando un microprocesador tiene un número limitado de contadores hardware, una aplicación de grandes proporciones puede requerir días y semanas de ejecución, tanto para reunir la suficiente información, como para realizar un análisis coherente del funcionamiento. El problema de la multiplexación viene a la hora de subdividir el uso de los contadores hardware; se hace a través del tiempo, por lo que vamos a ganar más información en detrimento del número de horas de ejecución.

Para realizar la multiplexación deberemos ejecutar el método `PAPI_multiplex_init()` y después de crear el conjunto de eventos, haremos un `PAPI_set_multiplex(EventSet)`.

Es conveniente no utilizar eventos derivados, ya que puede producir algunos errores a la hora de utilizar multiplexación.

La multiplexación tiene un gran inconveniente y es que no todas las plataformas la soportan. Otro problema a tener en cuenta es que la multiplexación provoca cierta sobrecarga cuando cambia de eventos. Además, ningún evento es medido durante el tiempo total de análisis. Estos factores pueden afectar a la precisión de los resultados asociados a los eventos.

En cuanto a la plataforma en la que estamos trabajando la utilización de la multiplexación fue en un principio una gran opción, ya que, si monitorizamos más de tres o cuatro eventos obtenemos resultados incoherentes con un orden grandísimo (véase el subapartado resultados incoherentes en el apartado 3.5 Errores), según la FAQ de PAPI, esto se resuelve mediante la multiplexación, pero a la hora de implementarla surgieron problemas. Al invocar el método `PAPI_set_multiplex` se obtenía un error -19 que ni siquiera está contemplado en la lista de errores arrojados por PAPI.

Después de este error el programa sigue ejecutando y a la hora de añadir más eventos obtenemos un error -8 (`PAPI_ECNFLCT`).

La FAQ dice que si obtenemos un error -8 deberemos utilizar `PAPI_set_multiplex` cada vez que vayamos a añadir un evento, pero en el caso de realizar esto para cada evento surgen dos nuevos problemas: `PAPI_set_multiplex` devuelve -1 (`Invalid argument`) en todas las ocasiones, y además todos los eventos tienen un valor de cero después de comenzar y leer, e incluso después de realizar ciertas tareas de computación.

Se adjunta junto con este documento un programa de prueba de la multiplexación en el que se intentaron solventar los problemas pero no se pudo, probablemente la plataforma no soporte la multiplexación. La única solución posible es implementar la multiplexación de forma manual. Para ello deberemos ejecutar las mismas pruebas varias veces, cambiando de eventos en cada ejecución.

### 3.5 Errores

Las librerías de PAPI poseen un conjunto de códigos de error preestablecidos, éstos se muestran en la siguiente tabla:

VALOR	SÍMBOLO	DEFINICIÓN
0	PAPI_OK	No hay error
-1	PAPI_EINVAL	Argumento invalido
-2	PAPI_ENOMEM	Memoria insuficiente
-3	PAPI_ESYS	Una llamada a una librería de C falló, comprobar error
-4	PAPI_EBSTR	La resta devuelve error, normalmente el resultado de una funcionalidad no implementada
-5	PAPI_ECLOST	El acceso a los contadores ha sido perdido o interrumpido
-6	PAPI_EBUG	Error interno, por favor envíe un correo a los desarrolladores
-7	PAPI_ENOEVT	El evento hardware no existe
-8	PAPI_ECNFLCT	El evento hardware existe, pero no puede ser monitorizado debido a limitaciones de hardware
-9	PAPI_ENOTRUN	Ningún conjunto de eventos se está ejecutando actualmente
-10	PAPI_EISRUN	El conjunto de eventos está actualmente ejecutando
-11	PAPI_ENOEVST	Conjunto de eventos no disponible
-12	PAPI_ENOTPRESET	No es un evento preestablecido válido
-13	PAPI_ENOCNTR	El hardware no soporta contadores hardware
-14	PAPI_EMISC	Error desconocido

#### Resultados Incoherentes

Otro de los errores con los que nos podemos encontrar, y suele ser bastante común, es que los contadores devuelvan valores normales cuando trabajo con uno o dos eventos, pero cuando añado alguno más recibo un error -8, PAPI\_ECNFLCT, y además los contadores toman valores exageradamente altos:

```
PAPI_TOT_INS = 577732409434650008
PAPI_INT_INS = 577756873568368160
PAPI_FP_INS = 577736708696913352
PAPI_TOT_CYC = 4479200
```

Código 14: Salida de un programa en el que usamos más contadores de los existentes

El problema es que hemos excedido el número de contadores hardware disponibles en la plataforma, una posible solución es implementar la multiplexación temporal, o en caso de que nuestra plataforma lo permita, utilizar la que proporciona PAPI.

## 4.- example.c

En los archivos finales se entrega el archivo example.c, en el que se implementa el problema del acceso a datos con stride (salto), esto es, accedemos a la matriz por columnas de tal manera que el programa cargará toda una página de memoria para sólo acceder a un simple número de ella. Cuanto mayor sea el número de filas menor será la localidad y provocará un mayor número de fallos en caché.

Este example.c recibirá el tamaño de la matriz y el número de filas que poseerá dicha matriz.

Además con el archivo script.sh, podemos realizar un barrido desde 1 fila, hasta el segundo parámetro introducido, aumentando de forma exponencial, es decir, 1 fila, 2 filas, 4 filas, así hasta N2.

El primer parámetro que se le introduzca al archivo script.sh será el número de elementos que poseerá la matriz (N), y que deberá ser múltiplo al número de filas. Por lo tanto,  $N2 > 0$  y  $N2 > N$  Y  $N\%N2 == 0$ .

En caso de que deseemos ejecutar una matriz que ocupe 1.5 MB, asumiendo que en la arquitectura en la que estamos tratando  $\text{sizeof(int)} == 4$  Bytes, el parámetro N deberá valer  $1.5 * 1024 * 1024 / 4 = 393216$ .

A continuación, mostramos la salida del script en caso de pasarle como parámetros una matriz de 1.5MB y 32 filas como máximo de filas.

```
./script.sh 393216 32
Rows: 1      Tiempo: 1233      L1DATAMISS: 24612689      L2_LINES_IN 418223
Rows: 2      Tiempo: 1688      L1DATAMISS: 24613618      L2_LINES_IN 193676
Rows: 4      Tiempo: 2189      L1DATAMISS: 30163189      L2_LINES_IN 180602
Rows: 8      Tiempo: 2473      L1DATAMISS: 28004340      L2_LINES_IN 293339
Rows: 16     Tiempo: 6567      L1DATAMISS: 413137145     L2_LINES_IN 410762
Rows: 32     Tiempo: 6794      L1DATAMISS: 575078102     L2_LINES_IN 4212022
```

Código 15: Salida del script.sh

## Apéndice

### CODIGO DE INTERFAZ DE ALTO NIVEL

```
/*compilado con gcc prueba_hl.c -o prueba_hl.o -lpapi*/
#include <papi.h>
#include <stdio.h>
#define NUM_EVENTS 3

main()
{
    int i = 0, j = 0;

    int Events[NUM_EVENTS] = {PAPI_TOT_INS, PAPI_FP_INS, PAPI_TOT_CYC};
    long_long values[NUM_EVENTS];
    char EventCodeStr[PAPI_MAX_STR_LEN];

    /* Start counting events */
    if (PAPI_start_counters(Events, NUM_EVENTS) != PAPI_OK) {
        printf("1-----error\n");
    }

    /* Read the counters */
    if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK) {
        printf("2-----error\n");
    }

    /*Print results*/
    printf("-->1 (Read, starting)\n");
    for(i=0; i<NUM_EVENTS; i++){
        PAPI_event_code_to_name(Events[i], EventCodeStr);
        printf("%s = %lld\n", EventCodeStr, values[i]);
    }

    /* Do some computation here */
    for(i=0; i < 1234567890; i++){
        j = j+i*23;
    }

    /* Read the counters */
    if (PAPI_read_counters(values, NUM_EVENTS) != PAPI_OK) {
        printf("2-----error\n");
    }

    printf("-->2 (Stop, end)\n");
    for(i=0; i<NUM_EVENTS; i++){
        PAPI_event_code_to_name(Events[i], EventCodeStr);
        printf("%s = %lld\n", EventCodeStr, values[i]);
    }

    /* Stop counting events */
    if (PAPI_stop_counters(values, NUM_EVENTS) != PAPI_OK) {
        printf("3-----error\n");
    }
}
```

**CODIGO DE INTERFAZ DE BAJO NIVEL**

```
/*Compilar con: gcc prueba_low_level.c -o prueba_low_level.o -lpapi*/
#include <papi.h>
#include <stdio.h>
#define NUM_EVENTS 2

main(){

    int i = 0;

    long_long values[NUM_EVENTS];

    int retval, EventSet=PAPI_NULL;

    char EventCodeStr[PAPI_MAX_STR_LEN];

    /* Initialize the PAPI library */
    retval = PAPI_library_init(PAPI_VER_CURRENT);
    if (retval != PAPI_VER_CURRENT) {
        printf("Error(%d): PAPI library init error!\n", retval);
    }

    /* Create the Event Set */
    retval = PAPI_create_eventset(&EventSet);
    if (retval != PAPI_OK) {
        printf("Error: PAPI_create_eventset\n");
    }

    retval = PAPI_add_event(EventSet, PAPI_L2_LDM);
    if (retval != PAPI_OK) {
        printf("Error (%d): PAPI_add_event 1\n", retval);
    }

    retval = PAPI_add_event(EventSet, PAPI_L2_STM);
    if (retval != PAPI_OK) {
        printf("Error (%d): PAPI_add_event 2\n", retval);
    }

    /* Start counting events in the Event Set */
    if (PAPI_start(EventSet) != PAPI_OK) {
    }

    /* Read the counting events in the Event Set */
    if (PAPI_read(EventSet, values) != PAPI_OK) {
    }

    printf("-->After reading the counters\n");
    for(i=0; i<NUM_EVENTS; i++){
        printf("valor %d = %lld\n", i, values[i]);
    }

    /* Reset the counting events in the Event Set */
    if (PAPI_reset(EventSet) != PAPI_OK) {
    }
}
```

---

---

PAPI: Guía de Instalación y uso

```
/* Add the counters in the Event Set */
if (PAPI_accum(EventSet, values) != PAPI_OK){
}

printf("-->After adding the counters\n");
for(i=0; i<NUM_EVENTS; i++){
    printf("valor %d = %lld\n", i, values[i]);
}

/* Stop the counting of events in the Event Set */
if (PAPI_stop(EventSet, values) != PAPI_OK){
}

printf("-->After stopping the counters\n");
for(i=0; i<NUM_EVENTS; i++){
    printf("valor %d = %lld\n", i, values[i]);
};

retval = PAPI_cleanup_eventset(EventSet);
if (retval != PAPI_OK){
    printf("Error (%d): PAPI_cleanup_eventset\n", retval);
}

} //end of main
```

---

## Apéndice B: Ejemplo de uso de Huge Pages

---

A continuación mostramos el código utilizado para hacer uso dentro de Huge Pages:

```
int shmid1;

shmid1 = shmget(2, MB_8, SHM_HUGETLB | IPC_CREAT | SHM_R | SHM_W);
if ( shmid1 < 0 ) {
    perror("shmget");
}
//printf("HugeTLB shmid: 0x%x\n", shmid1);
A = shmat(shmid1, 0, 0);
if (A == (int *)-1) {
    perror("Shared memory attach failure");
    mshmid1, IPC_RMID, NULL);
}
```

La función `shmget` es la clave dentro de este código. A continuación explicaremos el funcionamiento de esta.

```
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

La función `shmget` devuelve el identificador de memoria (compartida) asociada con el argumento `key`. Un identificador de memoria compartida, asocia una estructura de datos y un segmento de memoria compartida de al menos “size” de tamaño. Se crean para el argumento “key” si una de las siguientes condiciones es verdad:

- El argumento `key` es igual a `IPC_PRIVATE`.
- El argumento `key` no tiene ningún identificador de memoria compartida asociado con él y (`shmflg&IPC_CREAT`) no es cero.

Cuando el segmento de memoria compartida se crea, será inicializado a cero en todos sus valores.

En caso de error la función devuelve un entero negativo. Por otra parte, si todo ha funcionado correctamente devolverá el identificador de memoria (puntero).



## Apéndice C: Manual de usuario

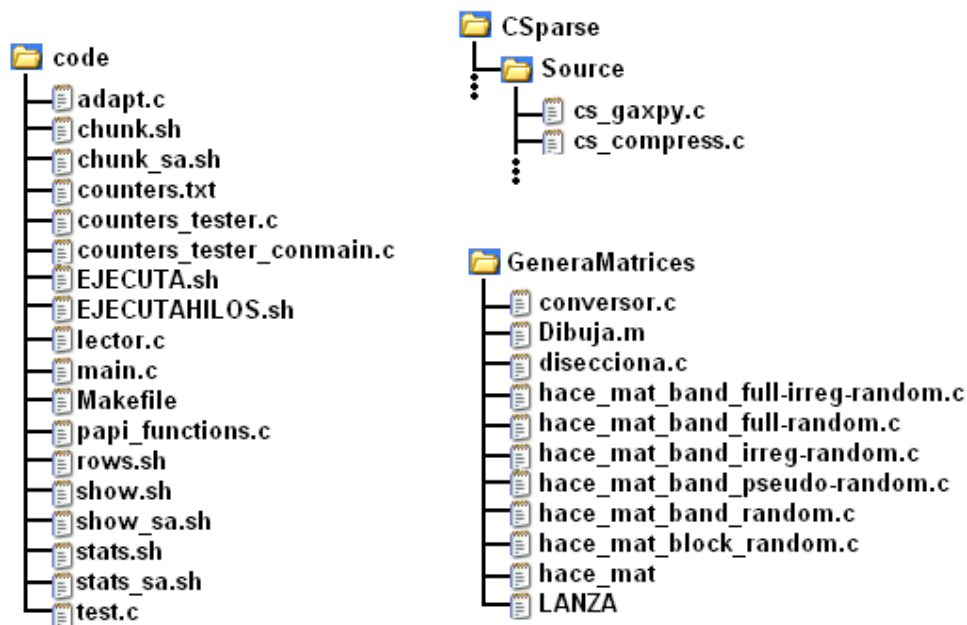
En este manual de usuario trataremos la estructura de archivos del benchmark, así como los pasos para ejecutar todo tipo de pruebas.

### 1. Descripción de archivos

Los archivos que se entregan como parte de este benchmark se pueden diferenciar en tres carpetas:

- **code:** carpeta donde se encuentra la parte principal del benchmark.
- **CSparse:** carpeta que contiene el código para la utilización del producto matriz-vector modificado. Esta carpeta se deberá encontrar al mismo nivel que la carpeta **code**.
- **GeneraMatrices:** en esta carpeta se incluye el generador de matrices, así como el conversor de matrices que hemos utilizado para que según se crean dichas matrices se vuelquen a un formato entendible por la función **load** de Tim Davis (CSparse).

Después de conocer la estructura de carpetas listamos los ficheros que se encuentran dentro de cada una de ellas explicando en qué consisten:



## Carpeta code

Esta carpeta contiene la parte principal del benchmark, vamos a ver en qué consisten los distintos ficheros, explicamos todos ellos por orden alfabético:

### **adapt.c**

Este archivo contiene todo lo concerniente a las dos técnicas de optimización de las que hemos hablado. Tanto Simulated Annealing como la Búsqueda N-aria. Al comienzo del fichero tenemos algunos de los parámetros de Simulated Annealing y más adelante los tres parámetros de la búsqueda N-aria, SECTIONS → número de secciones N, K y el número máximo de ejecuciones.

La función graphic es la que consigue a partir de unos puntos definidos formar una función que une estos puntos con rectas.

Después tenemos la función de cálculo de energía E1 que es llamada tanto por Simulated Annealing como por la Búsqueda N-aria. Dentro de ella hay una llamada sprintf, que se encarga de ejecutar el benchmark, y el resto de código hasta el final de la función se encarga de recuperar de la salida del benchmark el contador que deseemos (counter\_pos), que dependerá de los contadores que hayamos introducido en “counters.txt”.

Los siguientes métodos M1, S1 y P1 son las funciones de cálculo de diferencia de energías entre dos puntos, cálculo del siguiente salto e impresión de energía respectivamente.

La función narysearch() se corresponde con la Búsqueda N-aria.

Para finalizar tenemos la función main, que si llamamos con 0 ejecutará Simulated Annealing y si llamamos seguida de un 1 ejecutará la búsqueda N-aria.

### **chunk.sh**

Este script se encarga de llamar al método main pero con distintos tamaños de grano (chunk). El primer parámetro es la función del main a ejecutar, luego el número de elementos, el número de filas y el número máximo de filas que queremos ejecutar. Su progresión es exponencial en 2, es decir irá haciendo 1, 2, 4, 8, 16 así hasta NUM. El quinto parámetro es el número de hilos con el que queremos que se ejecute la función.

El while no hace más que ejecutar con los distintos tamaños de granos y hace diferencia de si estamos ejecutando un producto matriz-dispersa vector o no, esto es necesario puesto que para pasar el fichero tenemos que redirigir la entrada estándar.

Este script se encarga de llamar el script show.sh al final de la ejecución.

**chunk sa.sh**

Realiza la misma función que el script anterior pero sólo para un único tamaño de grano, que es el que se indica con NUM. Este script está orientado a Simulated Annealing y Búsqueda N-aria que sólo requieren de la ejecución de un único tamaño de grano.

**counters.txt**

Es el fichero de texto plano que contiene los contadores que queremos utilizar. Se pueden introducir comentarios en este archivo si van precedidos del carácter almohadilla '#'.

Los contadores deberán empezar una nueva línea e irán seguidos de retorno de carro.

**counters tester.c**

Este archivo contiene la librería que se encarga de tomar los contadores del anterior archivo y realizará pruebas hasta conseguir separar los contadores válidos para la arquitectura en la que estemos ejecutando. Y además agrupará tantos como pueda en una única prueba. Esta función almacena en un array que se pasa como entrada información sobre si los contadores son correctos para la arquitectura o no. Si el valor es negativo es que no son válidos y si es positivo indicará el número de prueba en la que se les incluirá.

**counter tester conmain.c**

Es el mismo código que el anterior pero este posee main por si el usuario decide comprobar previamente que contadores son válidos y cómo los puede agrupar para conseguir minimizar el número de ejecuciones. Para esto último sólo tendría que variar la posición de los contadores dentro del archivo counters.txt

Este fichero mostrará por consiguiente una salida por pantalla para indicar el resultado del test de contadores. Esta salida se puede comprobar dentro de la memoria, en el apartado 4.1.1 Selección dinámica de contadores.

**EJECUTA.sh**

Es un script que se utiliza para ejecutar como su propio nombre indica la función producto matriz-dispersa vector para las matrices creadas por el generador de matrices. Variando SIZE, DIAG Y EXPAND estaríamos variando las carpetas en las que entramos, que serán las que se hayan creado. Se recomienda ver el script LANZA que se encuentra en la carpeta GenerarMatrices para entender mejor el funcionamiento de éste.

Como parámetro de entrada requiere el número de hilos con los que queremos que se ejecute dicho script.

**EJECUTAHILOS.sh**

Script sencillo que se encarga de llamar al script EJECUTA.sh con distintos número de hilos.

**lector.c**

Programa creado para generar distintos tipos de estadísticas sobre un conjunto de valores de entrada. Para obtener la media y desviación típica le llamaremos utilizando los parámetros de entrada 4 0.

**main.c**

Es la parte principal del benchmark, se encarga de ejecutar las distintas pruebas y de inicializar los valores de las matrices.

En el comienzo de este fichero se inicializan las variables y se comprueba que el número de argumentos corresponde con el nombre de la función.

El siguiente paso será llamar a counters\_tester() para leer en el fichero counters.txt los contadores que el usuario quiere utilizar. La salida se almacena en counters\_info y de aquí sacaremos las distintas pruebas que realizaremos. Tomamos los contadores y los inicializamos.

Después viene la parte de reserva de memoria según las funciones que vayamos a ejecutar. Cabe destacar el caso del producto matriz-dispersa vector, que se encarga de llamar a las librerías de CSparse modificadas por nosotros. En dicha sección del manual explicaremos los archivos con los que se relaciona directamente.

El if que tiene como entrada la variable CLEANCACHE, se encarga como su propio nombre indica de borrar la caché. Previamente deberemos de indicar en la variable CACHESIZEMB el tamaño de la caché máxima o de la que se desee borrar.

Más adelante comenzamos a tomar estadísticas y arrancamos los contadores (start\_papi), después llamaremos a la función según el parámetro de entrada que se asigne a la función main.

Por último se muestran los valores de los contadores papi con la función print\_papi, y además también se imprimen por pantalla los valores de salida de la función, el tiempo total de ejecución y la salida de los resultados de borrar la caché. Todos ellos necesarios para que el compilador no elimine las partes de código que no producen alguna salida final.

En la parte final del código tenemos las salidas de la comprobación de errores de los parámetros de entrada.

**Makefile**

Es el encargado de compilar todo el código. Producirá cinco ejecutables:

- main: parte principal del benchmark que ejecutará las pruebas y funciones creadas.

- lector: programa que genera estadísticas
- conversor: programa para convertir matrices creadas por el generador de matrices o del grupo de Tim Davis, a la una versión entendible por la función load de las librerías CSparse.
- counters\_tester: evaluador de contadores a partir del fichero counters.txt.
- adapt: programa que sirve para llamar a las técnicas de optimización: Simulated Annealing y Búsqueda N-aria.

Además contiene algunos test para poder ejecutar de forma sencilla. Este Makefile también se encarga de llamar al Makefile que se encuentra dentro de la carpeta CSparse. Esta es otra de las razones por las que se recomienda que la carpeta CSparse se encuentre al mismo nivel code.

### **papi functions.c**

Este fichero contiene todas las llamadas a PAPI y la comprobación correspondiente de errores de contadores. El fichero main.c llama constantemente a funciones de papi\_function.c y se pasó a este archivo para simplificar el código de main.c.

Entre las funciones más destacadas encontramos papi\_start() para que los contadores comiencen la cuenta; papi\_stop() para que los contadores dejen de contar; read\_papi() para leer los valores de los contadores y print\_papi() para imprimir los valores.

### **rows.sh**

Realiza la misma función que chunk.sh pero para ejecutar el benchmark con distintos número de fila

### **show.sh**

Script que se encarga de mostrar los resultados arrojados por rows.sh y por chunk.sh. Sólo mostrará los datos si show es igual a 1. Además este script se encarga de guardar las estadísticas en ficheros para que luego el script de estadísticas las pueda recoger y llamar el lector con ellas. Esto último sólo lo realiza si sabe es igual a 1.

### **show\_sa.sh**

Realiza la misma función que show.sh pero solo muestra/guarda un tamaño de grano. Orientado a las técnicas de optimización.

### **stats.sh**

Script encargado de ejecutar varias veces la misma prueba. Los resultados que han sido previamente almacenados por show.sh (con save activado). Son recogidos y llamamos al lector.c para obtener la media y la desviación típica de los elementos

**stats\_sa.sh**

Misma función que stats.sh pero para un único tamaño de grano. Se usa en Simulated Annealing y Búsqueda Nária (se llama en la función de energía E1).

**tests.c**

Contiene las pruebas básicas del benchmark, entre ellas tenemos:

- matrix: Escribe en una matriz por filas. Escribe en memoria de forma consecutiva.
- matrix\_read: Lee de una matriz por filas. Lee de forma consecutiva.
- matrix\_reverse: La que hemos denominado en la memoria escritura con stride. Escribe en una matriz realizando saltos por tamaño de fila. Recorre la matriz por columnas.
- matrix\_reverse\_read: Igual que la anterior pero sólo en modo lectura, es decir, lee dando saltos una matriz previamente inicializada.
- false\_sharing: Prueba en la que varios procesos escriben en una misma zona de memoria.
- El resto de funciones provienen de la carga de matrices dispersas, ver “cs\_gaxpy.c”.

**Carpeta CSparse**

Se recomienda tomar la carpeta original y sustituir dos ficheros de la carpeta Source. Estos ficheros son:

**cs\_gaxpy.c**

Este archivo contiene el producto matriz-dispersa vector. Las funciones que podemos encontrar son:

- cs\_gaxpy: función original de las librerías de Tim Davis. Es el producto matriz-dispersa vector por columnas.
- cs\_gaxpy\_col: función paralelizada de la anterior, debido gran número de dependencias, al ejecutar de forma paralela ofrece distintos resultados.
- cs\_gaxpy\_cols2: función que denominamos de “array expansion”, cada hilo tiene su propio array donde guarda los resultados y al final de la ejecución se suman todos los resultados. En este caso el resultado final es el mismo para todos los casos.
- cs\_gaxpy\_rows: función que multiplica la matriz-dispersa y el vector por filas, lo que hace que cada hilo no comparta zonas de memoria idénticas, aunque en ocasiones compartan bloques.
- cs\_gaxpy\_rows2: esta función es similar a la anterior, pero en ella definimos el trabajo de cada hilo de forma manual.

- `cs_gaxpy_rows3`: prueba que se hizo para ver si aumentando los cálculos se conseguían poner al 100% ambos procesadores en el core2Duo. Después de añadir más cálculos ambos núcleos alcanzaban el máximo rendimiento.
- `cs_gaxpy_rows4`: cuarta función de multiplicación por filas. En este caso los datos se almacenan en una variable auxiliar sobre la que al final se le aplica el operador de OpenMP reduction. Lo que estamos haciendo es sumar en esta variable final el valor de todas las variables privadas de cada hilo. Esta prueba se hizo para ver la importancia de la carga y modificación del vector y, dentro de la computación de esta prueba. Los resultados que se obtienen es que hay un menor número de fallos y mejor funcionamiento en el sistema, pero las diferencias no fueron muy grandes, la parte más importante desde el punto de vista de carga de datos está en la matriz dispersa que posee muchos más elementos que el vector.
- `cs_gaxpy_row5`: esta función se hizo igual que la anterior para no cargar los valores de la matriz, y cargarlos del vector. En este caso se ven los tiempos reducidos a gran escala.

### **cs\_compress.c**

En este archivo es donde se produce la compresión de una matriz dada en formato triplete a una matriz en formato csc. En nuestro caso tuvimos que añadir el método `cs_compres_csr` que se llama desde la función `main` y que es el encargado de realizar la compresión de la matriz por filas, para poder ser utilizada por las funciones `cs_gaxpy_row`.

## **Carpeta GeneraMatrices**

El directorio `GeneraMatrices` posee el código creado para la creación de matrices con distintos grados de dispersión. El fichero de ejecución principal es `LANZA`. Éste se encarga de generar las matrices para los parámetros de entrada:

- `SIZE`. Indica el tamaño de filas de la matriz. Las matrices creadas siempre son de estructura cuadrada, luego también se corresponde con el número de columnas.
- `DIAG`. Indica el número de elementos medio que existirán en cada fila.
- `EXPAND`. Refleja la dispersión de los elementos dentro de la fila. Siendo el valor 1 una matriz dispersa en la cual todos sus elementos no cero se encuentran concentrados en la diagonal y el valor 100 una matriz en la que los elementos no cero se encuentran repartidos de forma aleatoria por toda la fila.

Este script se encarga de llamar al programa `hace_mat_band_full-irreg-random` con los tres parámetros, y este se encarga de generar la matriz. El programa disecciona separa la matriz en el fichero `columnas` y `filas`.

Para finalizar nuestro `conversor.c` se encargará de transformarla a un formato entendible por la función de carga de la librería `CSparse`.

Este fichero conversor tiene distintas opciones, si introducimos como parámetro 0 se encargará de transformar una matriz del conjunto de la Universidad de Florida al formato triplete entendible por la función de carga. Y si por el contrario introducimos 1 o 2 traducirá del formato generado por la antigua versión o la nueva del generador de matrices al formato deseado respectivamente.

Además en `Dibuja.m` se incluye código para que las matrices sean representadas gráficamente mediante Matlab.



## 2. Pruebas

Para finalizar este apéndice veremos distintos ejemplos de ejecución de pruebas, para la ejecución del benchmark. Dentro de los ficheros se dejan pruebas de ejemplo para que el usuario que desee comience a probar el benchmark. En la parte final mostraremos el funcionamiento de los otros ejecutables que se incluyen en la práctica.

### [Escritura con stride](#)

La prueba que vemos a continuación es con una matriz de 786432 elementos, que equivalen a un total de 3MBytes =  $3 \cdot 1024 \cdot 1024 \cdot 4$ . Este último 4 es por el tamaño del entero. El segundo parámetro corresponde con el número de filas de la matriz, en este caso 16

```
./main matrix_reverse 786432 16
```

### [False sharing](#)

En este caso tenemos cuatro parámetros: número de elementos de la matriz, número de filas (prácticamente despreciable ya que se recorre de forma continua), el tamaño de grano y el número de hilos con los que deseamos ejecutar la prueba

```
./main false_sharing 393216 4 256 2
```

### [Producto matriz-dispersa vector por filas](#)

Los dos primeros parámetros se ignoran, el tercero es el tamaño de grano y el cuarto el número de hilos de la matriz. Finalmente se redirige la entrada estándar al fichero que representa la matriz que multiplicaremos por el vector.

```
./main matrix_vector_product_rows 0 0 512 12 < matrizdeentrada.mtx
```

### [Script chunk.sh y rows.sh](#)

Los parámetros se mantienen, lo único que varía es que se iniciará la cuenta en tamaño de grano/número de filas 1 hasta el indicado por el tercer parámetro.

```
./chunk.sh false sharing 393216 4 256 2
```

### [Script stats.sh](#)

Lo único que cambia al ejecutar es que se añaden dos parámetros más al anterior, uno será el script a ejecutar y será el primer parámetro y el último será el número de repeticiones que queremos realizar sobre esa prueba.

./stats.sh row.sh matrix\_reverse 786432 0 256 0 5 (se ignora en esta prueba el número de hilos ya que siempre se ejecuta con uno solo).

```
./stats.sh chunk.sh matrix_vector_product_rows_0 0 256 2 30 ../CSparse/Matrix/
```

**NOTA ADVERTENCIA:** En ocasiones dependiendo de la arquitectura puede que si el script tiene el principio del fichero `#!/bin/bash` no funcione o sí. Se recomienda quitar/añadir esta sentencia para conseguir que funcionen los scripts.

### Técnicas de optimización

Para ejecutar las técnicas de optimización simplemente deberemos llamar al ejecutable `adapt` seguido de un 0 si queremos ejecutar Simulated Annealing, o seguido de un 1 si queremos ejecutar la Búsqueda N-aria.

`./adapt 0` o `./adapt 1`

Si queremos cambiar la prueba que se ejecuta dentro deberemos de modificar dentro de `adapt.c` la función de energía `E1`. En la sentencia `sprintf` se llama al benchmark y en las posteriores se recoge la información. Cambiando el valor de la variable “`counter_pos`” recibiremos la salida de un contador u otro. Para modificar las distintas opciones de las técnicas de optimización como el tamaño de salto de Simulated Annealing, o las variables `K` y `N` de la Búsqueda N-aria, tendremos que modificar los valores definidos como constantes al principio del fichero. Para modificar en Simulated Annealing dónde comenzaremos a ejecutar el algoritmo deberemos cambiar el valor de la variable “`x_initial`”.

### Counters tester

Para ejecutar el testeador de contadores simplemente escribiremos el nombre del ejecutable seguido del fichero del cual deseamos comprobar los contadores.

`./counters_tester ficheroaanalizar`

### Lector

Para obtener los valores en medios y la desviación típica en el lector llamaremos de la siguiente forma al ejecutable:

`./lector fichero de entrada 4 0`

En el fichero de entrada se los valores deben ir separados por retornos de carro.

### Conversor

El conversor se realizó para que las matrices se preparasen para cargar con la función de carga `cs_load` de las librerías de Tim Davis. Según el primer parámetro le estamos indicando al programa que tipo de matriz de entrada tiene, estos valores pueden ser:

- 0: Cuando convertimos de la colección de Tim Davis. Se encarga de borrar los comentarios y restar 1 a todas las coordenadas. Esto último lo hace puesto que las matrices son 1based, el mínimo valor es 1. Sin embargo, la función de multiplicación soporta el 0.
- 1: Cuando convertimos de la antigua versión del generador de matrices.
- 2: Cuando convertimos de la nueva versión del generador de matrices.

Por tanto la llamada para convertir una matriz de Tim Davis al correspondiente formato sería:

```
./conversor 0 ficheromatrizentrada ficheromatrizsalida
```